1978

Memory management policies for a hardware implemented computer operating system

William A. Kwinn
Iowa State University

INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR

Memory management policies for a hardware implemented computer operating system

by

William A. Kwinn

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1978

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Virtual Memory and Dynamic Memory Management

Two features of computer systems developed to expand the scope of practically addressable problems are virtual memory (15) and dynamic memory management (8). The purpose of a virtual memory system is to provide the programmer with the illusion of working with a practically unbounded memory. This effect is achieved by a combination of a small, random access memory called core, and a large, much slower backing store. The backing store is usually a rotating magnetic device, such as a drum or fixed head disk. The backing store is organized into a large number of storage blocks of fixed size called pages. The core is organized into a much smaller number of elements called core frames, each capable of holding one page. In the core, individual words may be addressed, whereas in the backing store, only whole pages may be accessed. Among systems, pages range in size from approximately one hundred to several thousand words.

A program may execute only if copies of the instructions and data it needs are in core. A reference made to an item not in core is called a page fault. When a page fault occurs, a mechanism (either software or hardware) is invoked to bring the necessary page into core memory, and if necessary, to write the replaced page to the backing store. In order that the relationship between the core memory and the backing store remains invisible to the programmer, the executing program references items only by the addresses of their backing store

locations. A hardware unit, such as an associative memory, intercepts every.memory reference, replacing the backing store page address with the appropriate core frame number whenever the referenced page is in core, and generating a page fault interrupt whenever it is not.

In virtual memory systems, programs too large to fit in core may be written without concern for either memory size or explicit overlay considerations. They are found to run with acceptable performance if they enjoy locality of reference and do not suffer from scattering. Locality of reference is the name given to the phenomenon that within a short (but not trivial) period of time, a program tends to reference a small subset of the total number of words in its memory space. The set of pages upon which this subset lies will be called a locale. Locality of reference is a property of a program determined by the algorithm, code, and data structures used. The size of a locale is determined by the manner in which the system upon which the program is implemented maps the program code and data into memory space. Programs which develop large locales are said to display scattering.

Dynamic memory management is a facility provided by a language implementation to allocate and deallocate variable amounts of space as needed during program execution. Older languages such as FORTRAN provide no such facility. Languages such as ALGOL, PASCAL, and PL/I provide for the allocation and deallocation of memory at block entry and exit times. PASCAL and PL/I further allow the programmer to allocate and free memory explicitly. APL, LISP, and SNOBOL are widely known languages which provide implicit dynamic memory management. In

all of these languages, this facility is provided by software. The SYMBOL programming language, described briefly in this paper, provides implicit dynamic memory management which is implemented directly in hardware.

When storage is managed implicitly, appropriate memory space is associated with a name when a value is first assigned to that name during execution. If subsequent assignments to this name produce different memory requirements, the system automatically obtains whatever additional storage is necessary. When space is no longer needed, it may be reclaimed for later reuse. (The method by which obsolete storage is identified and reclaimed is highly dependent upon the implementation details of the individual system.) Implicit management thus frees the programmer from the tedious bookkeeping details which accompany declarations and explicit management of storage and from the inflexibility of static typing.

Performance in Virtual Memory Systems

Systems which successfully implement both virtual memory and dynamic memory management must be designed to avoid harmful interactions of the two features. In a virtual memory system, the cost of performing a memory reference is very much higher when the reference causes a page fault than when it does not. A greater than necessary number of page faults may occur if the dynamic memory management system is not designed to minimize the number of pages involved in a given allocation. Since the SYMBOL computer system

(19,20,21,24) has a virtual memory and provides highly dynamic memory management, it is a natural vehicle for the investigation of the interactions of these features. The research reported here was undertaken to evaluate the performance of SYMBOL according to the measures of execution time and number of page faults and to propose alterations which would reduce harmful interactions in a SYMBOL-like system.

Two factors which strongly impact the performance of a virtual memory system are the quality of its page replacement algorithm and the degree of scattering exhibited by the programs. A number of page replacement algorithms have been studied by several investigators. Among the simpler ones are random replacement and first-in first-out (22), which make no use of the history of past page references. The least recently used (22), working set (9), and page fault frequency (6) algorithms do incorporate a short term reference history, and have been found to be generally more effective than the simpler algorithms. Algorithms which incorporate a longer history of reference might be superior, however no page replacement algorithm can significantly improve a program's performance if the program is badly scattered.

A program which references many distinct pages in short intervals of time displays the phenomenon of scattering. The degradation of performance becomes serious when the number of pages referenced is consistently greater than the number of core frames available to the program. There are several identifiable causes of scattering. One cause is the use of instructions which produce branches to remote

sections of code. Such branches may be the result of explicit GO TOs, implicit jumps in other control constructs, the invocation of ON blocks, and procedure calls. Parameter passing and global variable handling mechanisms which require repeated access to a procedure's calling environment may also aggravate scattering. This is particularly true of call by substitution, with its deferred evaluation of expressions. Another potential cause for scattering is poor placement of data. Scattering of data references occurs when a program rapidly accesses many separate items on different pages or when it references a large data object which occupies portions of many pages. Systems which allow dynamic memory management are particularly susceptible to data scattering, especially if data values can dynamically change size and shape or contain pointer values or shared data.

When memory references are sufficiently scattered, it is likely that only a few words on a page will be referenced between the time the page is brought into core and the time it is removed to make room for another page. Thus the ratio of referenced to unreferenced items on a page is small. Attempts to reduce scattering center on increasing this use ratio.

If the scattering involves the program code, it may be possible to improve performance by restructuring the code. Program restructuring attempts to increase the use ratio by causing highly interrelated sections of code to occupy a small set of pages (11,12,14). Program restructuring is usually accomplished by making

repeated runs of the program and arranging that for each subsequent compilation, sections of code which have been determined to be needed together will be placed together. The recognition of such sections of code, if left to the programmer, is dependent upon his experience and intuition. Another possibility is to rely upon instrumentation such as address traces to determine how program blocks interact and to identify the sections of code to be rearranged. It is conceivable that intelligent compilers may be able to assume this task without incorporating information obtained on previous runs, but this has not been demonstrated. Experimental results (14) indicate that program restructuring can be more effective in improving a program's performance than changing the page replacement algorithm. Unfortunately, program restructuring has certain limitations. First, data dependencies may thwart its effectiveness. In addition, it can only be applied to code and data which are known at compile time and are static. Restructuring offers no help to the scattering caused by dynamically varying data.

One technique for increasing the use ratio which applies equally well to code and data, whether static or dynamic, is to reduce the page size (4a,5b,13b). If memory references are severely scattered, then the adoption of a smaller page size can significantly increase the use ratio and reduce the number of page faults because core frame space that would have been occupied by the unreferenced portions of large pages will instead be available for other pages containing referenced items. Thus, the use of a larger number of smaller pages

operating in the same amount of core permits a richer variety of locales to become core resident during the execution of a program. If the references are only mildly scattered, then smaller pages may leave the use ratio practically unchanged. In this case, more paging operations will be required to bring the same amount of information into core. The potential advantages of using smaller pages are opposed by the cost in time of paging operations and the cost in mapping hardware proportional to the number of core frames.

Another approach to increasing the use ratio is to adopt a memory allocation strategy which attempts to minimize data scattering. For languages which allow declarations, it might be reasonable to allow the programmer to specify related data items in the declarations. The compiler could then use this information to allocate compact storage for the related items. In systems which use no declarations, the memory allocator would need to be aware of the affinity of an allocation request for existing data, the page size, and how much space is available on a page in order to try to allocate space compactly (4c).

The Problem

Experience has shown that both dynamic memory management and virtual memory reduce the programmer's burden. It would be desirable, then, to obtain the beneficial effects of these features in a single system. Unfortunately, dynamic memory management and virtual memory implementations can interact to degrade performance because the

scattering of memory references can be aggravated by dynamic allocation of storage, causing a large number of page faults during program execution. The purpose of the research presented in this paper is to demonstrate techniques which reduce the harmful interactions of these two features. The techniques suggested were successfully implemented on the SYMBOL-2R computer system.

One technique for performance enhancement presented in this paper is the use of small pages. Data will be presented which show that the use of small pages improves performance of SYMBOL programs whose memory demands far exceed the size of core memory. (These data are presented in terms of the number of page faults encountered per job execution rather than in terms of the more traditional measures of page fault frequency (5b) or lifetime functions (4b), which relate page faults and virtual time. Such measures are not directly obtainable for SYMBOL programs because of the nonuniform virtual time requirements of SYMBOL's operators and memory service requests. Results which are presented in terms of page fault frequencies differ from those presented for SYMBOL only by a multiplicative constant, the total virtual time elapsed during program execution. Lifetime functions are simply multiplicative inverses of page fault frequency functions.)

Another technique studied here is a memory allocation strategy which attempts to cause a structure to grow onto pages it already occupies. This strategy was found to be very effective in reducing the number of page faults incurred by a program. The results presented here extend the work done on LISP (4c) to a more ALGOL-like

language and show that improvements are available at practically no cost.

The data taken demonstrate that significant improvements in SYMBOL's performance can be obtained either by the use of smaller pages or by the adoption of a sophisticated, yet easily implemented, memory allocation strategy. It is believed that the results reported here transcend the implementation on SYMBOL and that the techniques studied should be considered in the design or modification of other virtual memory systems which support dynamic memory management.

THE SYMBOL SYSTEM

Origins and Architecture

The experiments discussed in this paper were performed on the SYMBOL-2R computer, a time-sharing system which was designed and built by Fairchild Camera and Instrument Corporation Digital Systems Research Division. The only prototype ever built was obtained by Iowa State University, where research on its unique architecture has been sponsored by the National Science Foundation.

The design of the SYMBOL computer began with the specification of the language to be supported by the hardware. The language specification incorporated a variety of features, some of the more significant of which are outlined below. The language was to be block-oriented to facilitate functional partitioning of programs and to limit the scope of variables. There was to be no cryptic "job control language" nor any burdensome declarations. The data values were to be able to vary in size and shape dynamically. After specification, this language was directly implemented in hardware.

The hardware to support the SYMBOL language was partitioned into eight relatively autonomous processors. Their names and a brief description of their functions may be found in Table 1, and a conceptual representation of their interconnections may be found in Figure 1.

All of the processors, except the Channel and Drum Controllers, communicate with the Memory Controller via a common bidirectional bus

Table 1. SYMBOL's hardware processors

Name	Function
Memory Controller(MC)	Provide all memory service, manage dynamic memory linking and virtual memory mapping
Memory Reclaimer(MR)	Convert deleted storage into storage available for reallocation
Interface Processor(IP)	Exchange information between core buffers and virtual memory space and support hardware loading and editing of program source
Translator(TR)	Produce from program source text a string of postfix object code and a set of name tables
Central Processor(CP)	Execute programs using the code produced by the translator
Job Controller(JC)	Schedule the MR, IP, TR, and CP and handle page replacement for the virtual memory
Channel Controller(CC)	Exchange information between the "outside world" (terminals) and the core buffers
Drum Controller(DC)	Shuttle pages of memory between core and drum

which includes fields for address, data, terminal number, memory op code, and page list number. All requests for memory service are made over this bus and are handled by the Memory Controller according to the fixed priority of the requesting processor. All processors exchange status bits and completion codes with the JC via dedicated

unidirectional lines. (In practice, these lines are multiplexed on the main data bus when such use does not conflict with memory requests.)

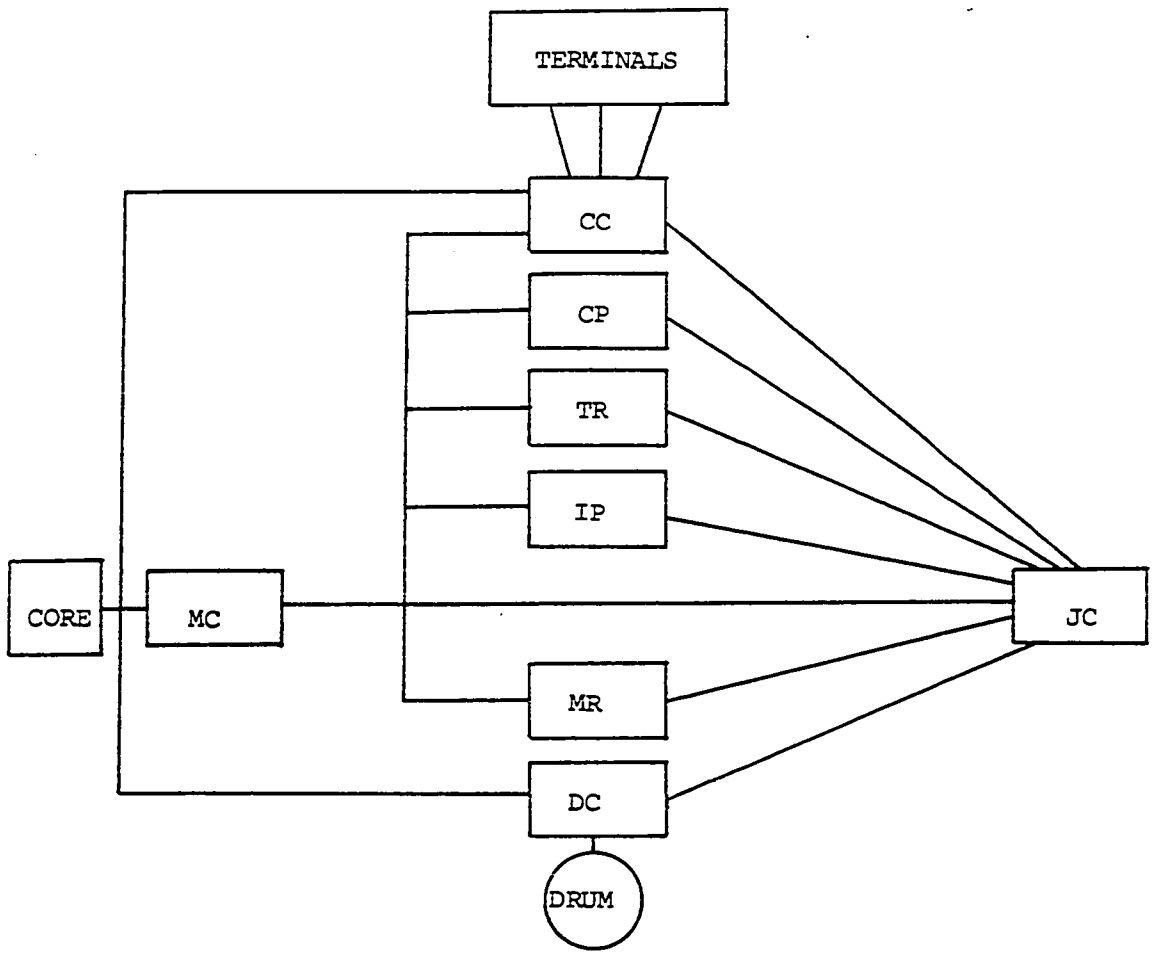


Figure 1. Conceptual data paths in SYMBOL

The Memory Subsystem

In order to understand the experiments which have been performed, it is first necessary to understand SYMBOL's virtual memory configuration and storage allocation and deallocation techniques. SYMBOL's memory consists of a 32 page, 5 usec core memory and a 4096 page drum with a 16 msec rotational period, organized with four pages per track. Each page consists of 256 sixty-four-bit words. Up to 28 pages of SYMBOL core are used as a virtual memory buffer. The remaining four pages of core are reserved for system tables, terminal status save areas, and I/O buffers. They are forced to be resident, and hence, are not available to the virtual memory. These four pages are accessible via "direct" memory operations which bypass the virtual memory mapping. The rest of core, although similarly accessible, is manipulated only through the "virtual" memory operations.

Each of the virtual memory pages is made up of a data storage region and a 32 word overhead region. The data storage region is organized into 28 eight-word groups. In SYMBOL, the group, and not the word, is the quantum of allocation. Each group in the data storage region is associated with a unique group-link word in the 32 word overhead region. A group-link word contains two addresses which are the forward and backward pointers to the next and previous groups of a logical string. The remaining four words in the overhead area are used to maintain lists of the users' pages, to store information about the available space on a page, and to store information for the page replacement algorithm.

The virtual memory demand paging algorithm is executed by the Job Controller. The JC maintains a linked list of the pages in core and associates with each such page a two-bit value which indicates that page's resistance to being replaced. When a memory request by an active process cannot be satisfied by the pages in core, the JC determines the two-bit priority of the request and records the required page address, which processor requested the missing page, and for which terminal that processor was acting. The JC must select a core frame to fill with the requested page. It scans the list of pages in core and selects the first page it encounters having a resistance to replacement less than the priority of the request. If there is no such page, the JC selects the first page whose resistance value is equal to the request's priority. If the JC cannot select a page for replacement during the first request, the requesting processor will be allowed to repeat its request at each subsequent request time until the request can be granted. Once the page to be replaced has been selected, the replacement is made, and the newly arrived page is inserted at the rear of the JC's list and its resistance value is determined. CP requests cause a resistance value of 10, TR and IP requests cause a value of 01, and others cause a value of 00. This resistance value is increased by one if the job which caused the request is at the head of the CP queue or if the request has been issued by the TR for a name table page. The JC uses the same basic rule for assigning priorities to memory requests, with the exception that a priority is increased by one only if the

requesting job is at the head of its processor's queue. When only one terminal is running on the CP, the page replacement algorithm degenerates to a first-in, first-out scheme.

The processors in SYMBOL view memory as a collection of arbitrary length strings. Each string may consist of a sequence of physically adjacent or nonadjacent substrings. The primitive operations available to the hardware processors in SYMBOL are more complex than mere "reads" or "writes." For instance, when the Central Processor wants to create a new string, it requests that space be allocated to it by the Memory Controller. The MC returns to the CP the address at which the space begins. The CP must save this address, in a register, a nametable, or on the execution stack, as appropriate, so that the CP can later request that the storage be deleted. The CP can store data into successive words of this string, implicitly allocating whatever space is needed. It may fetch successive words of this string in either direction.

When a processor requests memory service, it specifies a particular Memory Controller operation (See Table 2.), and, depending upon the particular operation, it may provide an address with or without data. Upon completion of the MC operation, the requesting processor will have nothing returned, or an address with or without data will be returned.

Any group in SYMBOL's memory is in one of three states, available, allocated, or garbage. Transitions from one state to another occur cyclically, as shown in Figure 2. The Memory Controller

Table 2. Memory Controller operations

Operation	123456[a]	Function
Assign group(AG)	ox o	Obtain the beginning address of a new string.
Insert group(IG)	xx o	Obtain a new group and insert it between the specified group and its successor.
Fetch and follow(FF)	xxt x	Read the specified word and return the address of the successor word.
Fetch reverse(FR)	xxr x	Read the predecessor word and return its address.
Fetch direct(FD)	x t x	Read the specified physical core word, bypassing the virtual mapping.
Follow and fetch(FL)	xxr x	Read the successor word and return its address.
Delete to end(DE)	x x	Delete all successors of the specified word.
Delete string(DS)	x x	Delete entire string. Address transmitted must be the beginning of a string.
Store and assign(SA)	xxtx o	Write data transmitted in specified address. Return successor address. Allocate new group as successor if none existed.

[a]
1 - Address transmitted
2 - Address returned
3 - Address used internally
4 - Data transmitted
5 - Data returned
6 - Page list transmitted

x - Required
o - Optional
t - Transmitted
r - Returned

Table 2. (continued)

Operation	123456[a]	Function
Store direct(SD)	x tx	Write data transmitted in physical core address, bypassing the virtual mapping.
Store only(SO)	x tx	Write data transmitted in the specified address.
Store and insert(SI)	xxtx	Write data transmitted in specified address. When the group is exhausted, allocate a new group and link it into the string at that point.
Reclaim group(RG)	x x	If the address transmitted is 0, fetch the top of the terminal's garbage stack; if not 0, link the group onto its page's available group list.

is the only processor which is aware of the data structures that indicate which groups are available and which are garbage. When an allocation request is made, the MC selects a group, removes it from the pool of available groups, and relinquishes all control over it. When storage is deleted, it is placed on a garbage stack. The Memory Reclaimer directs the MC to take groups from the garbage pool and make them available. The allocation and deletion strategies of the Memory Controller are elaborated below.

Whenever space is needed, the space allocation mechanisms are directed by the intended use for this space. The three classes of memory usage are source program text, object string, and name tables

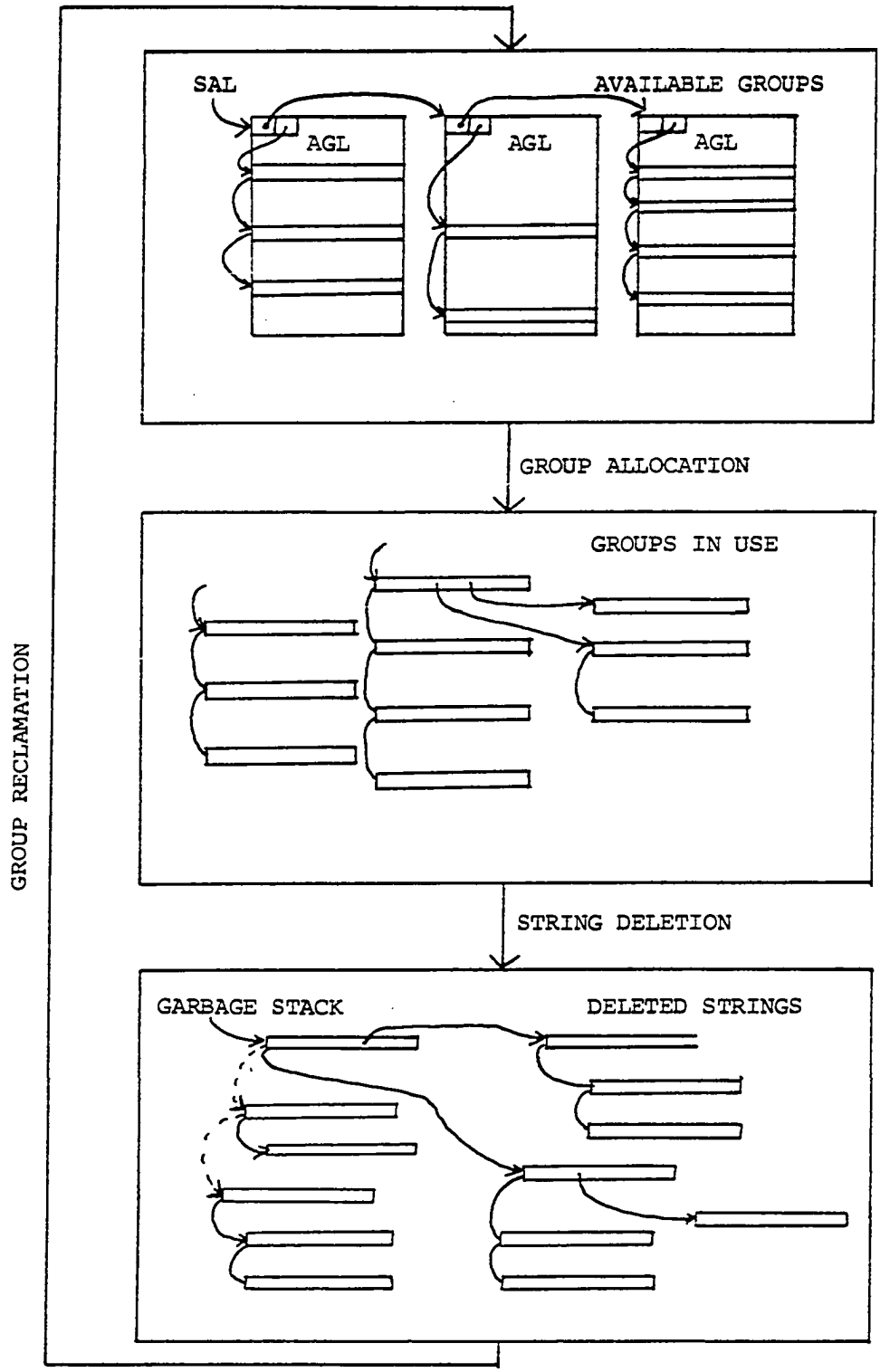


Figure 2. Life cycle of a group

and data. When a page is first allocated to a particular user from the system's pool of available pages, the space on that page becomes restricted to one of the three usage classes. All pages belonging to one user and having a common usage class are linked together on a list, called a page list. An entire page list can be cleared and its pages returned to the system pool with a single invocation of the Memory Reclaimer. This segregation of memory pages by usage class has a beneficial effect on program translation and execution times because of the compactness of the name tables. Since memory is allocated one group at a time and since memory can be reclaimed during a program's execution, it is common for a page to contain both allocated and available groups. All pages on a given page list which contain available groups are linked in a sublist of the page list, called its Space Available List (SAL).

Among the facilities provided by the primitive operations of the Memory Controller is the ability to allocate groups. An allocation is satisfied by the first available group which the Memory Controller finds. The search for space begins on the page specified in the allocation request, and then proceeds, if necessary, to the first page on the SAL of the specified page list for the requesting terminal. If the page which is on top of the SAL no longer has any available groups, it is removed from the SAL and the new top page is examined. If the SAL is exhausted without satisfying the request, a completely clean page is obtained from the system's pool of available pages and linked onto the appropriate page list and SAL.

There are two mechanisms which indicate the presence of available groups on a page -- a counter and a pointer. The counter is used in preference to the pointer whenever possible. It is used from the first allocation on a page until every group on the page has been allocated exactly once. When a page is first obtained from the system pool, a field in the page overhead region, called the initial group assignment counter, is initialized to the offset of the first group of the data region of the page. The counter thus indicates the first group on a page which has not been previously allocated. When an allocation is made, the address of the group indicated by the counter is returned to the requesting processor and the initial group assignment counter is incremented to point to the next group. When the last group on a page has been allocated, the initial group assignment counter is deactivated for the lifetime of the program. The second mechanism, a pointer, indicates the head of a linked list of all those groups in the data storage region on that page which have been reclaimed and hence are available for reallocation. When the pointer is used to find an available group, the group at the head of the available group list is removed from the list and its address is returned to the requesting processor. The pointer is then updated to point to the new head of the list.

Another facility provided by the primitive operations of the Memory Controller is the deletion of strings. Whereas a string is allocated incrementally by single groups, an entire string may be deleted with a single operation. When a string is deleted, it is

placed on the top of the garbage stack for the appropriate page list of the requesting terminal. A dedicated hardware processor, the Memory Reclaimer (MR), continuously polls all of the garbage stacks. When it encounters a nonempty stack, the MR successively inserts each group of the top string into the available group list of that group's page. Upon finding an explicit substructure pointer in a word, the MR deletes the string pointed to and destroys the pointer.

The SYMBOL Language

In designing the language to be supported on the SYMBOL system, the implementers wished to relieve the user of the burden of declarations. Beyond this, they wished to allow the type, size, and shape of a value to be dynamically variable. The scalar values in SYMBOL, whether arithmetic or character, may be of arbitrary length. SYMBOL's language allows the building of structures which are vectors whose elements are scalars and other structures. Both substructures and elements of structures are accessed via subscript lists. Any element of a structure, or the entire structure itself, may be replaced by any scalar or by a structure of arbitrary size and shape at any time. The short example below illustrates SYMBOL'S structure handling capabilities.

```
(1)  x|The value of X|;
(2)  q<9|5|Hello<Goodbye|7>>;
(3)  output q[3],q[4,1],q[4];
(4)  q[2] ← <x|19>;
(5)  q[4] ← ||;
(6)  output q;
```

In statement 1, x is given as an initial value the scalar

string 'The value of X'. In statement 2, q is given as an initial value the structure whose first, second, and third elements are the scalars '9', '5', and 'Hello', respectively, and whose fourth element is a structure whose two elements are the scalars 'Goodbye' and '7'. Statement 3 causes three lines of output, namely

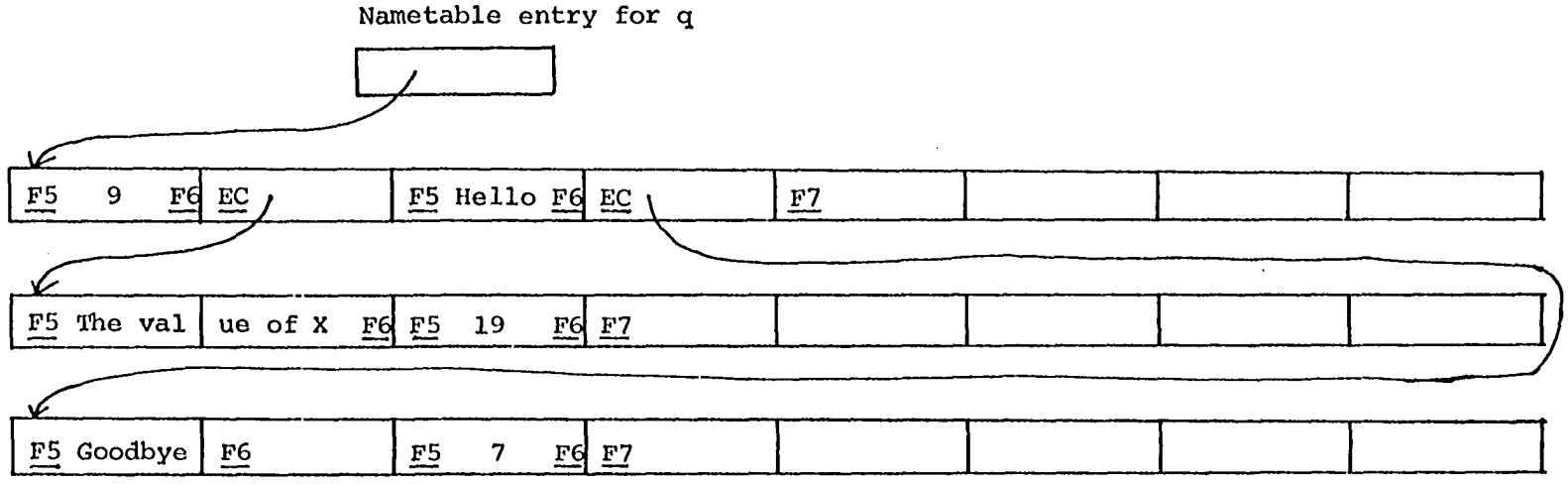
```
Hello
Goodbye
<Goodbye|7>
```

Statement 4 replaces the second element of q by a structure, and statement 5 replaces the fourth element of q by a null. Statement 6 causes the output shown below, which occupies two lines because the output formatter attempts to paragraph structures to enhance readability.

```
<9<The value of X|19>
Hello|>
```

The value of the structure created by line 4 of the program is depicted in Figure 3. Each eight-word group is shown on a single line. Several nonprintable internal characters are indicated by their underlined hex codes. F5 indicates the beginning of a character value, F6 the end of a character value, F7 the end of a vector, and EC a pointer to a substructure.

Because memory is allocated one group at a time and successive allocations may be on different pages, SYMBOL cannot use address arithmetic to resolve subscripted references. To resolve a reference to q[i], the Central Processor starts scanning at the beginning of the string q, counting components until it reaches the i-th component.



<9<The value of X|19>Hello<Goodbye|7>>

Figure 3. Internal representation of a structure

Likewise, to resolve a reference to q[i,j], the CP first finds the i-th component, as above, then finds the j-th component of the string pointed to by q[i]. Any number of subscripts may be used. In an attempt to reduce the amount of scanning needed to locate successive components of a structure, the following technique is used. When a reference to q[i,j] is resolved, i and a pointer to q[i] are stored in the nametable entry for q, and j and a pointer to q[i,j] are stored in the same word which contains the substructure pointer to q[i]. A subsequent reference to q[k] for k greater than or equal to i or to q[i,m] for m greater than or equal to j can then be resolved by following the pointers to q[i] or q[i,j], as appropriate, and beginning the sequential scan at that point.

The preceding short example illustrates the principles of the SYMBOL language which are necessary for an understanding of this work. The SYMBOL language is detailed further in (19).

THE EVALUATION PROCESS

Procedures and Equipment

A common approach to the study of virtual memory systems is to obtain and analyze a trace of the address references made during program execution (3,6,11,12,13a,14). In the SYMBOL system, this would require capturing three bytes of address per reference. References may potentially occur every five microseconds. Unfortunately, the hardware needed to intercept and record such a trace in real time is prohibitively expensive. Also, trace information cannot be extracted from SYMBOL at reduced speed without perturbing the interactions of the controllers involved with memory management and paging. Furthermore, it is not possible to obtain such an address trace via software since this would greatly perturb the memory reclaiming timing and would greatly change the paging traffic by virtue of its need to share the core with the program being traced.

An alternative approach was chosen for the study of SYMBOL's virtual memory system. The approach used was to vary memory management parameters and to record the occurrence of certain significant events which would allow the characterization of system performance. The events chosen were the completion of tasks, paging operations, and the execution of individual instructions. Since paging operations require a minimum of four milliseconds, a modest microcomputer can easily monitor this activity. In addition, measures of more rapidly occurring activity, such as the number of instructions

executed, can be provided by a few hardware counters read at page fault time.

The data presented in this paper were taken by running a set of experiments on the SYMBOL-2R computer. Each experiment consisted of a number of repetitions of the following activities. First the system was brought to a repeatable initial condition. Next various system parameters were set. One of a set of test programs was executed. During the program's execution, an external microcomputer monitored a selected set of processor activities. After execution terminated, the microcomputer processed and displayed the data it had taken. The variations performed on the system parameters are outlined below. A discussion of the possible implications of these variations and the actual experimental results appear in a subsequent section.

The number of core frames which were available for use as the virtual memory buffer was reduced from its maximum of 28 to as few as four frames. This was accomplished by forcing all used pages to be written to the drum, clearing core, and then explicitly building the linked in-core list which the Job Controller subsequently used to select core frames to fill.

Modifications to the Memory Controller hardware were made to allow the page size to be effectively reduced by restricting the number of groups in the data storage region which could be allocated. The number of allocatable groups was varied from 28 to two by externally forcing the value set in each page's initial group assignment counter at the time the page was acquired from the system's

pool of available pages. Setting this value caused the memory allocation hardware to ignore a corresponding number of groups on each page. Unfortunately, when the page size was reduced, the absolute core buffer size was also reduced since it was impossible to correspondingly repartition the core into a larger number of smaller page frames.

The algorithm for allocating space was independently modified to incorporate an additional, externally specified parameter, called the SAL threshold. This threshold specifies the minimum number of groups on a page which must be available in order to link that page onto the Space Available List (SAL).

Measurements concerning the dynamic behavior of program execution were obtained by the use of a small, external hardware monitor. This monitor consisted of a microcomputer interfaced to a set of counters, a CRT terminal, and a line printer. The microcomputer used was a MOS Technology, 6502-based KIM-1 with an additional 16K bytes of memory. Two styles of counters were used. Four eight-bit TTL counters capable of being cleared and read by the microcoputer counted high speed signals such as those indicating instruction fetches. TTL gates and buffers were used to interface these counters to the microprocessor. Preliminary measurements indicated a need for timing information and additional counters. To meet this need with the minimum amount of additional logic, three MOS Technology 6522 Versatile Interface Adapters (VIA) were added to the system. Each VIA contained a sixteen bit timer, a sixteen bit counter, two I/O ports, and all the logic

necessary for handshaking with an external device and for communication with the microprocessor's bus. The timers provided an accurate measurement of the time elapsed during program execution. The counters were used to record such slow activities as paging operations. Several of the I/O lines were used to monitor whether a processor was active.

The program for handling the dynamic measurement data was written in TINY BASIC (17) with speed-sensitive sections of code implemented as machine code subroutines. Output statements were used to mark the completion of designated portions of the SYMBOL programs being measured. At each instance of output activity in the SYMBOL program, the elapsed time and the cumulative totals of paging and Central Processor operation occurrences were recorded in the microcomputer's memory. At each page fault in the SYMBOL program, the number N of CP operations fetched since the previous page fault was used to compute the index I of a table T of values according to the following rule:

```
if    0 <= N <   16 then I ← N
if   16 <= N <  256 then I ← N/8    + 14
if  256 <= N < 8192 then I ← N/256  + 45
if 8192 <= N        then I ← N/2048 + 73
```

This function was chosen to provide fine resolution for small values of N, while reducing table T to a manageable size. The value in T(I) was then incremented by one, thus forming a distribution of the frequency of occurrence of a given number of CP instruction fetches between page faults.

Static measurements concerning the use of SYMBOL's memory and the

scattering of data values were made using programs run on a second SYMBOL terminal. The measuring programs were run in a privileged mode which enabled them to examine any memory space, including that belonging to another terminal. Data were taken by the measuring programs only when the measured programs were paused. These data were not perturbed by the measuring process since SYMBOL's memory allocation mechanisms implicitly allocate space to a program only on pages owned by the specific program making the request and since the measuring programs made no attempt to modify the values of the measured programs' pages. The measuring programs are described below.

The procedure NAME TABLE PAGESCAN viewed the data space of the measured programs as a collection of values and provided information about the scattering which occurred within structures of a SYMBOL program. It scanned all the values known in a program, printing the number of words and number of pages involved with each. It also determined the number of page transitions necessary to traverse each structure. The information provided by NAME TABLE PAGESCAN was valuable in the study of the behavior of SYMBOL programs because it told how many pages needed to be in core to access an entire value. The number of page transitions needed to scan the entire structure served as an indicator of the number of page transitions necessary to locate any of its substructures.

The program CONNECTIVITY was written to characterize scattering within a program's data space viewed as a collection of pages. It built a matrix C such that C(I,J) was the number of pointers from

items within page I to addresses within page J. This matrix was used to compute the number of on-page and off-page references and the percentage of off-page references for each page in the measured program's data space. In addition, the average percentage of off-page references was calculated.

The Sample Programs

The SYMBOL programs upon which the measurements were taken were selected from programs written over a period of several years by programmers of varying experience. They were chosen for their diversity of style, size, and purpose in an attempt to obtain a sample set representative of a typical SYMBOL workload. These programs were modified as necessary to eliminate the need for input by supplying the data as initial values in the source code. This was done to remove the random delays inherent in responding to prompts for input. Such delays can change the memory configuration because the MR can continue to perform memory reclamation during this time, and hence make the experiments not repeatable. Since the occurrence of output is an easily observed phenomenon, the amount of output produced by each program was limited to about a dozen lines, strategically placed to mark the completion of certain segments of the program. Each program was supplied with data sufficient to cause its execution time to be in the range of ten seconds to five minutes, an interval long enough to be nontrivial and yet short enough to make repeated runs feasible. A brief description of each measured SYMBOL program follows. A summary

of the gross characteristics of each is given in Table 3.

CROSSTEST is a character manipulating program which builds an alphabetical cross reference of all the identifiers in a procedure. It manipulates one extremely large and dynamic data structure.

ERGO finds all cycles (ergodic subchains) in an n x n transition matrix. In the process, it builds a vector of long character strings. Both the matrix and vector remain static after they are created.

IRVMOD2 is a character manipulating program which builds one very large array and several smaller ones. It searches for identities among the expressions in a nonassociative algebra.

KWIC does character manipulation to generate a KWIC index given a set of authors and titles. It manipulates one large structure which is fairly static after its incremental creation.

LEAST SQUARES is a numerical program which does matrix computations to find the best fitting polynomial of a given degree through a specified number of points. It creates and processes several matrices.

LIFE is a program to produce a pattern of characters on a cathode ray tube screen and to change the pattern according to certain population growth and decay rules. It manipulates two large static data structures.

Table 3. Characteristics of the sample programs

	CROSSTEST	ERGO	IRVMOD2	KWIC	LEAST SQUARES	LIFE
blocks	25	6	11	17	6	11
words of name table	741	166	390	331	231	207
words of object string	5248	1056	2824	1544	984	1456
CP operations performed (thousands)	816	886	4318	776	215	846
distinct operators	118	60	75	80	69	59
operator occurrences	2920	774	1627	1023	694	820
distinct operands	596	101	287	272	120	203
operand occurrences	2374	610	1369	793	508	715

RESULTS AND DISCUSSION

Two classes of experiments were performed on the SYMBOL-2R computer to determine whether simple modifications to the SYMBOL hardware could significantly enhance its performance. These experiments were motivated by the belief, later confirmed by the experimental results, that SYMBOL's memory references are severely scattered. The first class varied the page size and core memory configuration to investigate the improvements which could be obtained by a better use ratio. The second class introduced a new parameter, the SAL threshold, into the memory allocation strategy and measured the benefits provided by its use.

The Memory Size and Configuration Experiments

The first class of experiments was intended to determine the effect of core memory size and page size on performance. Each of the programs was run as the only active task in the system. Prior to each run, the system was brought to a fixed initial condition and the amount of available core memory was limited by specifying the number of core frames and the number of groups per page to be used. For each program, these two parameters were varied in three ways. First, the number of core frames used was decreased while holding the number of groups per page constant at the maximum value of twenty-eight. Next, the number of groups per page was decreased while using the full twenty-eight core frames. Finally, series of runs were made varying both the number of core frames and the number of groups per page, but

holding their product constant.

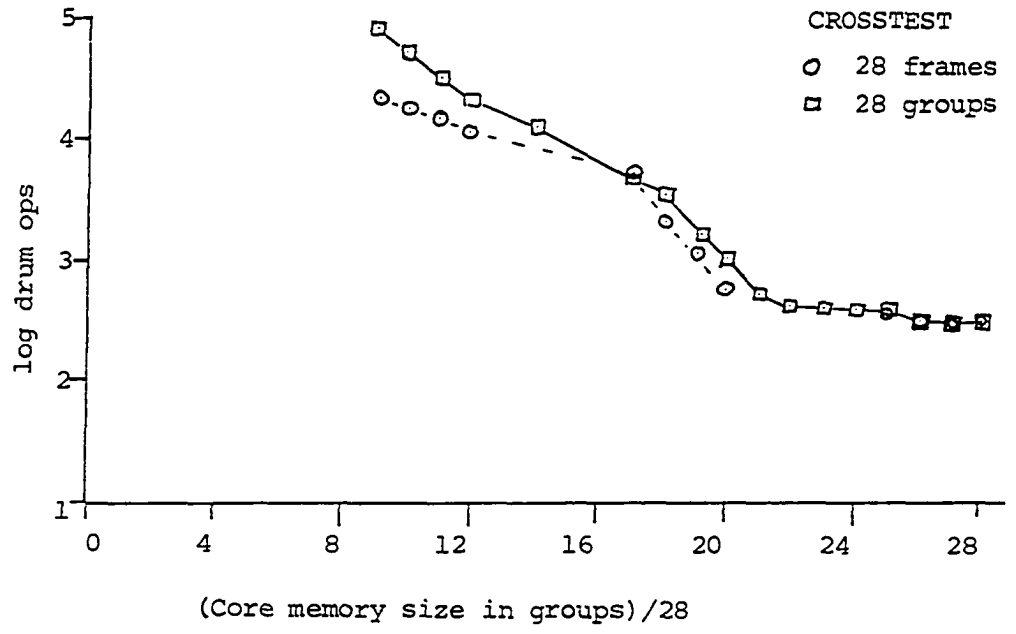
During each execution of the test programs, the microprocessor system monitored four signals on SYMBOL indicating the activity of the executing program. These four signals were CP active, I/O active, drum task active, and CP instruction fetch. The drum task active signal was counted, yielding the number of page transfers. The CP active signal was used to calculate the CP utilization and to determine the amount of work done between CP shutdowns. The signal was sampled twenty times per second, and a software counter was incremented each time the CP was busy. Another counter was incremented each time the signal was sampled. The ratio of these two counter values indicated the CP utilization. Each time the CP active signal fell, the count of the number of CP instructions fetched since the previous CP shutdown was used to build the distribution discussed earlier. Reading the count of CP instruction fetches after CP shutdown guaranteed that the counter was stable. The number of CP instructions fetched is the best obtainable indicator of forward progress in SYMBOL due to the fact that each CP instruction can cause the execution of many primitive memory operations, and if a CP instruction is interrupted, some subset of the memory operations already executed may need to be repeated when execution resumes. The I/O active signal was used to indicate the execution of output statements which were used as mileposts in the test programs.

For each test program, two curves are presented on a common set of axes. (See Figure 4.) These two curves contain the data taken in

the first two series of experiments. The horizontal axis gives the memory size and the vertical axis gives the common logarithm of the number of drum operations performed during execution. The logarithm was used in order to plot widely varying numbers on a reasonably small graph. The points joined by a solid line represent memory configurations having a constant number of groups per page. For this curve, the horizontal axis gives the number of core frames used. The points joined by a broken line represent configurations having a constant number of core frames. For this curve, the horizontal axis gives the number of groups per pages. For all of the programs except LEAST SQUARES, the constant used is the maximum value, twenty-eight. Since the memory requirements of LEAST SQUARES are so small, a more meaningful graph is obtained by using a constant value of twelve for both curves.

On each graph, both curves tend to be reverse S-shaped. For large memory sizes, the graphs are nearly horizontal. As the amount of memory decreases, the number of drum operations performed increases rapidly at first, and then more slowly until the program is no longer able to run. Although the general shape of the two curves for each program is dominated by the memory size, the increase in drum operations for decreasing memory size is far more severe for large pages. These graphs clearly show that it is not only memory size, but also memory configuration, which affects performance.

The third series of runs for each program was used to evaluate the effect of page size on performance, independent of memory size.



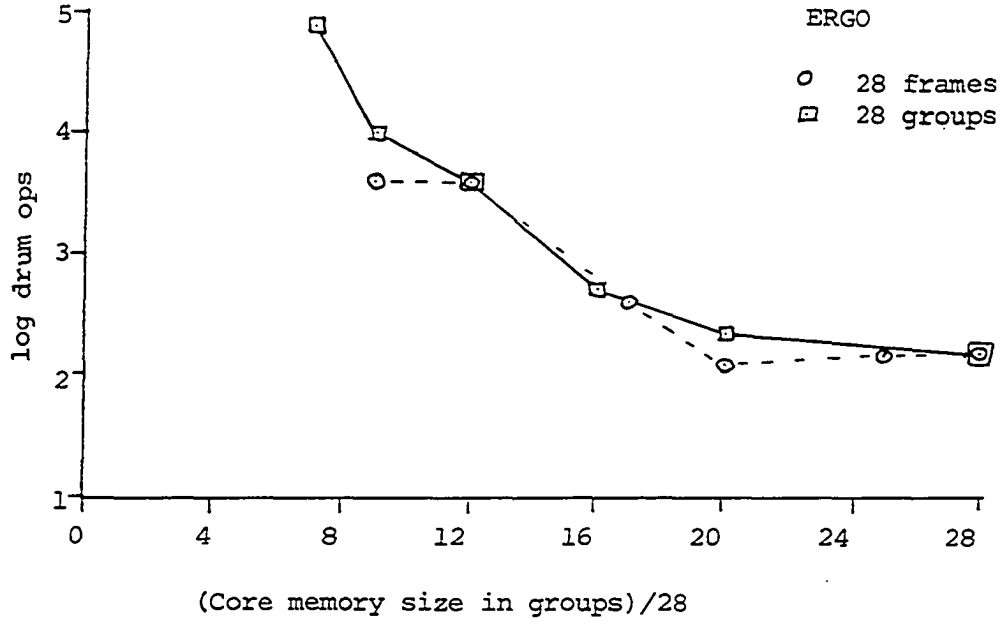
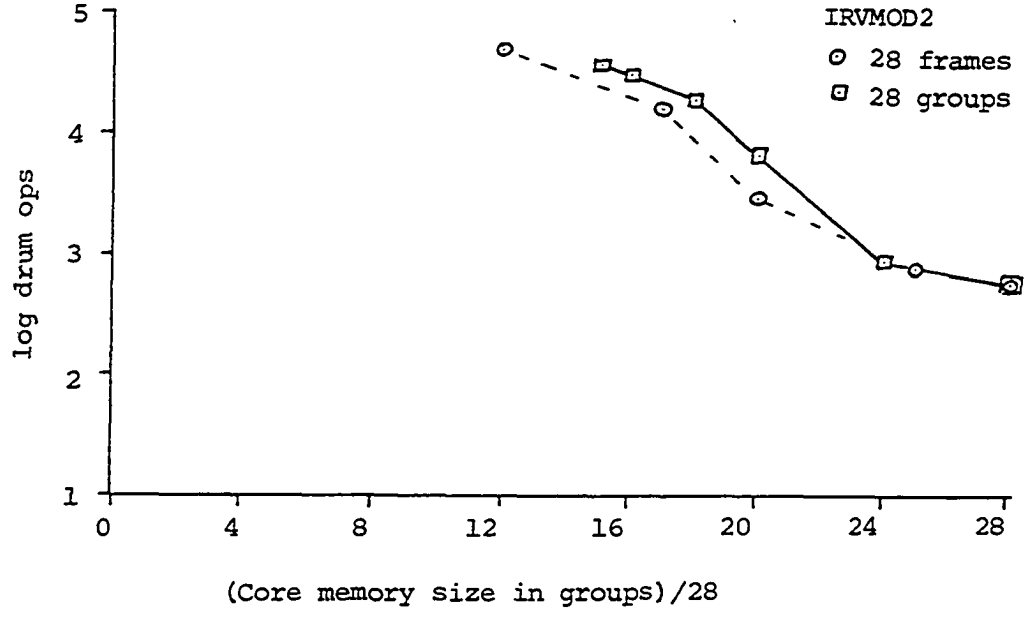


Figure 4. The effect of core memory configuration on drum operations with number of core frames or page size constant



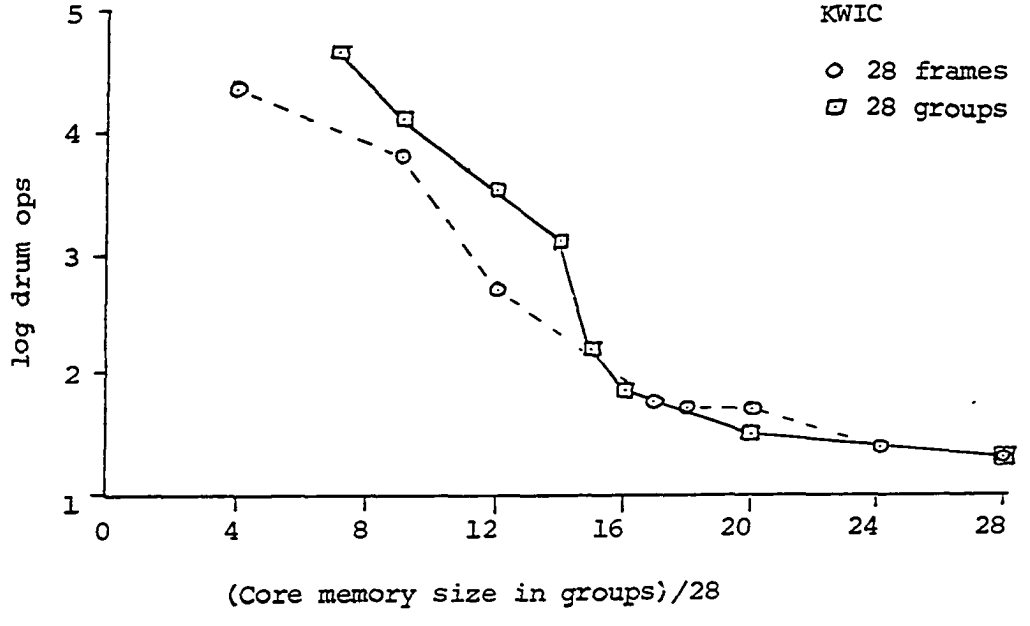
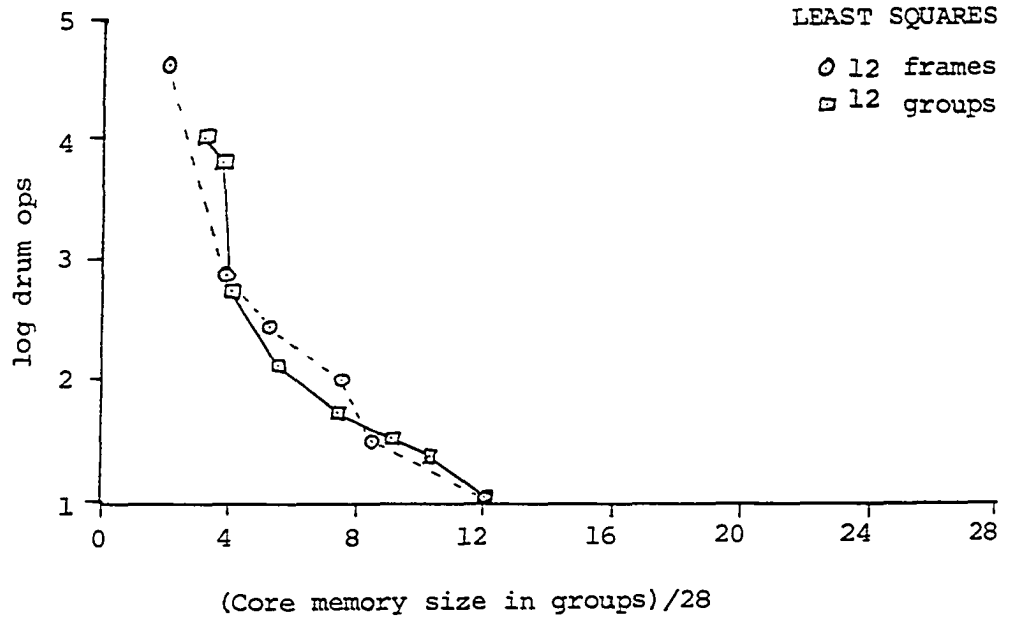


Figure 4. (continued)



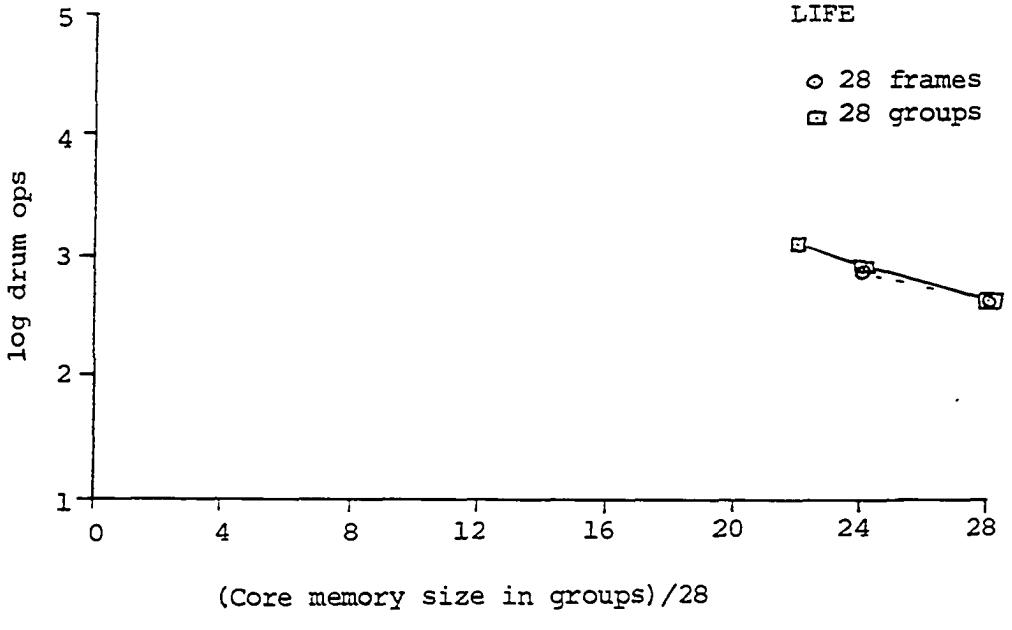


Figure 4. (continued)

The limitations of the hardware needed to specify the number of groups per page coupled with the large memory requirements of the programs IRVMOD2 and LIFE prohibited the gathering of any constant memory size performance data on these two programs. The data obtained are displayed in Table 4 and Figure 5. The quantities displayed in the table are the percentage of CP utilization, the execution time in seconds, the number of drum operations performed, and three percentages from the distribution of the number of CP instructions fetched between CP shutdowns -- the percentages of shutdowns having more than 0, more than 256, and more than 1024 operations fetched since the prior shutdown. The data displayed demonstrate that the amount of paging is significantly reduced by the use of smaller pages whenever the size of core is small enough to cause substantial paging. This reduction in the amount of paging is accompanied by an increase in the CP utilization.

The following examples illustrate the effects of page size on system performance. The program CROSSTEST thrashed to the extent that it would not run given nine pages of twenty groups each because the data which was needed to resume the execution of interrupted CP instructions was scattered over more than nine pages. This same program, however, was able to run when given the same amount of memory configured as twenty pages of nine groups each. Another program, KWIC, did run using both nine pages of twenty-eight groups and twenty-eight pages of nine groups, but using the smaller pages it performed only about half as many drum operations (6,612) as it did

Table 4. Constant memory size data

CROSSTEST	Memory size constant at 252 groups			
9,28	14,18	21,12	28,9	pages,groups
2.8	6.6	8.3	10.0	% CP busy
1151.1	492.2	388.4	325.5	seconds
90345	36668	28561	23610	drum ops
61.0	73.9	80.1	80.9	more than 0
0.5	1.5	2.1	3.0	more than 256
0.1	0.3	0.4	0.5	more than 1024

CROSSTEST	Memory size constant at 180 groups			
10,18	15,12	18,10	20,9	pages,groups
0.7	4.2	5.0	5.0	% CP busy
4809.3	775.0	639.5	646.0	seconds
134418	59606	48436	48870	drum ops
58.6	71.5	76.7	78.3	more than 0
0.5	1.1	1.2	1.5	more than 256
0.1	0.1	0.2	0.2	more than 1024

ERGO	Memory size constant at 252 groups			
9,28	14,18	21,12	28,9	pages,groups
29.9	27.5	33.3	47.1	% CP busy
167.8	182.4	150.6	106.4	seconds
9875	10848	8187	4697	drum ops
64.1	73.9	75.6	83.9	more than 0
9.9	7.4	9.9	24.2	more than 256
1.5	1.5	2.1	5.6	more than 1024

Table 4. (continued)

KWIC		Memory size constant at 252 groups		
9,28	14,18	21,12	28,9	pages,groups
18.1	22.3	27.3	30.0	% CP busy
193.2	157.3	128.2	116.7	seconds
13020	9776	7465	6612	drum ops
73.4	75.5	77.8	79.6	more than 0
10.4	12.9	16.2	15.1	more than 256
2.3	3.0	4.3	5.4	more than 1024

LEAST SQUARES		Memory size constant at 180 groups		
9,20	10,18	15,12	18,10	20,9	pages,groups
78.0	82.1	86.8	85.2	86.0	% CP busy
11.8	11.2	10.6	10.8	10.7	seconds
220	144	79	88	82	drum ops
71.2	68.7	79.6	82.0	84.5	more than 0
13.6	15.7	29.6	27.9	27.6	more than 256
5.1	9.6	13.0	13.1	17.2	more than 1024

LEAST SQUARES		Memory size constant at 108 groups		
6,18	9,12	12,9	27,4	pages,groups
12.0	10.9	49.7	74.2	% CP busy
76.6	84.0	18.5	12.4	seconds
5164	5901	790	217	drum ops
64.2	87.0	77.2	77.7	more than 0
4.6	6.7	19.9	16.9	more than 256
0.9	0.1	10.6	7.7	more than 1024

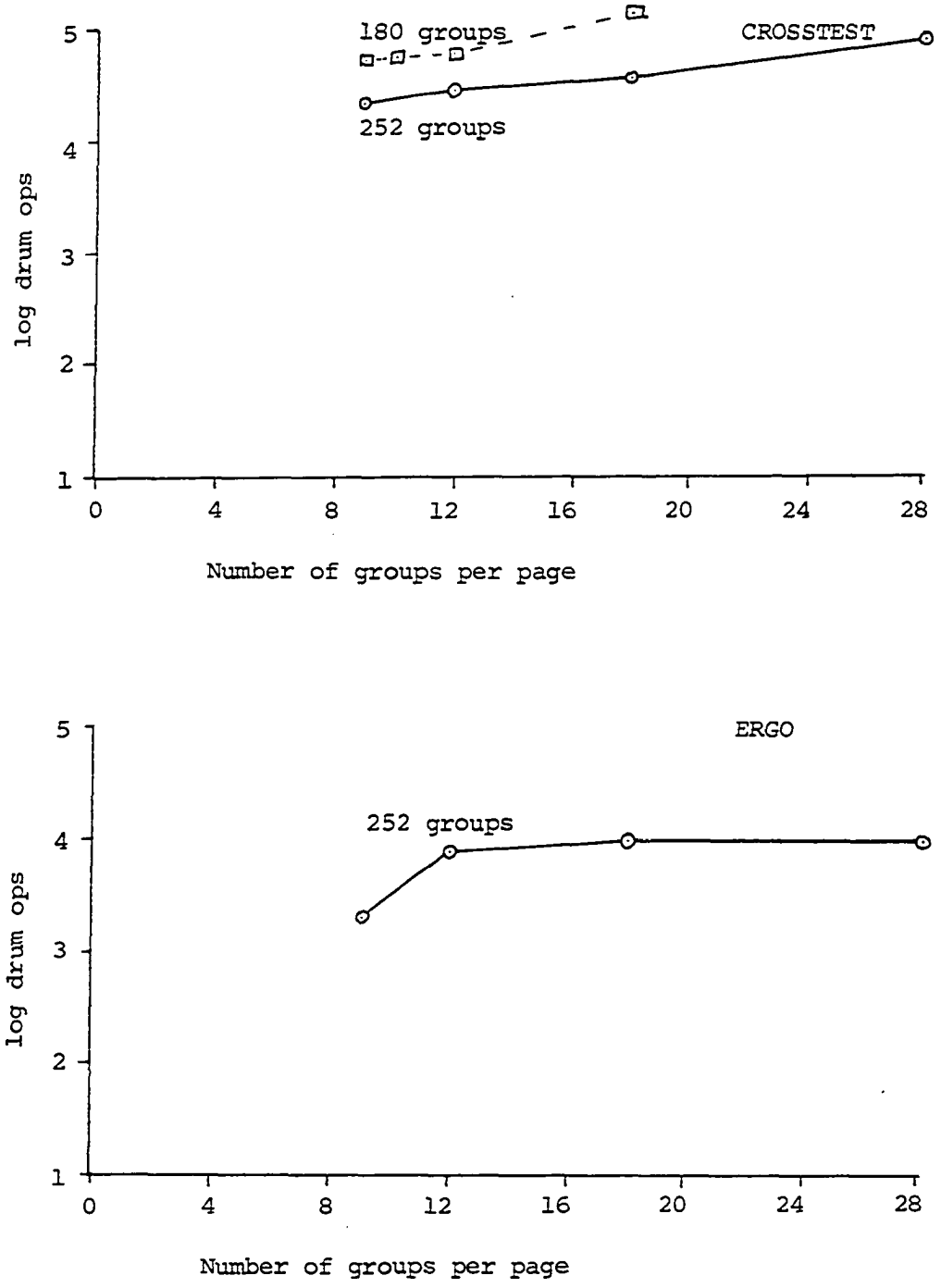
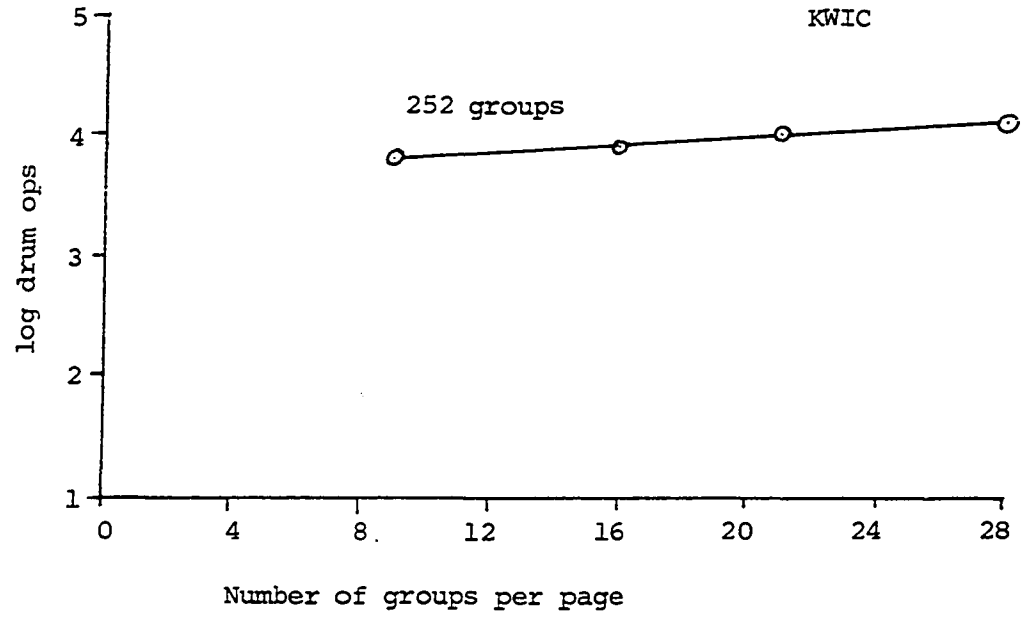


Figure 5. The effect of core memory configuration on drum operations with core memory size held constant



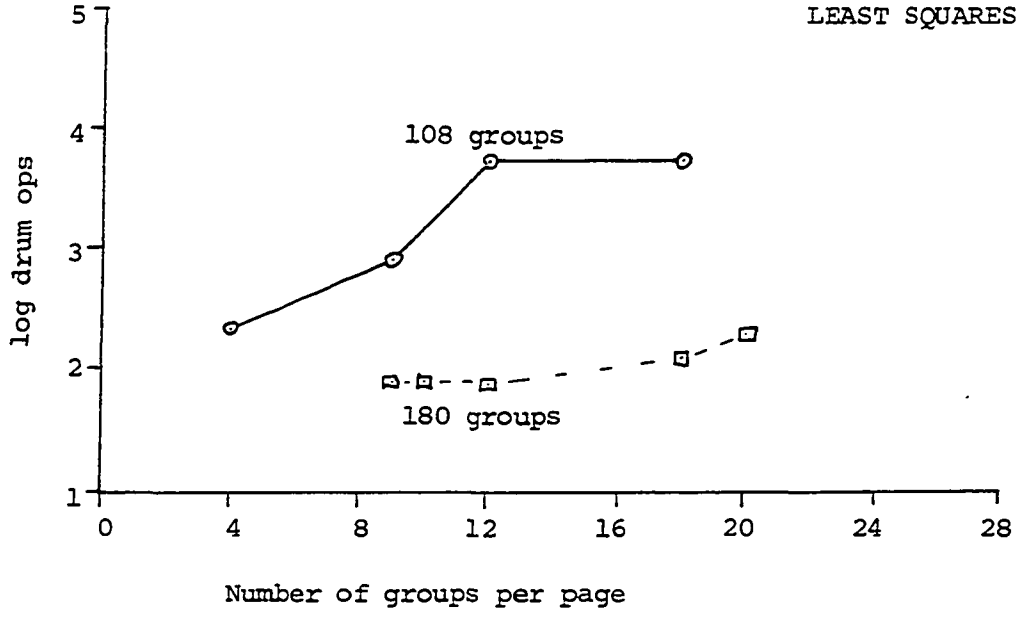


Figure 5. (continued)

when using the larger pages (13,020). The program ERGO obtained 47.1% CP utilization running with twenty-eight pages of nine groups. To obtain approximately the same utilization (50.0%) using twenty-eight group pages, a total of twelve pages, and hence one third more memory, was required. When KWIC was given a list of 100 titles and authors to index, instead of the list of ten which had been used above, it achieved a CP utilization of 9.3% at 9 pages of 28 groups each while doing 245,527 drum operations. At 28 pages of 9 groups each, it achieved 22.6% utilization while doing only 85,358 drum operations.

These data show that the use ratios for the tested programs have in fact been increased by the use of smaller pages and suggest that in general, smaller pages would be an improvement. The data covering page sizes in the range of nine to twenty-eight groups indicate that smaller pages always give better performance. Unfortunately, since it is not possible to increase the number of SYMBOL's core frames to accommodate the decrease in the number of groups per page, tests with fewer than nine groups per page are in most cases not feasible because the total amount of core memory is too small to permit execution. For the LEAST SQUARES program, however, twenty-seven pages of four groups do outperform twelve pages of nine groups. The data suggest that page sizes in the range of 100 to 1000 bytes may be quite suitable for systems, such as those supporting dynamic memory management, whose memory references tend to be highly scattered. As explained in the introduction, programs which exhibit little scattering will probably experience better performance on systems having the traditional,

larger pages.

When deciding on a page size for a virtual system, performance criteria may suggest the use of small pages, but additional factors may favor fewer, but larger, pages. Each core frame requires mapping hardware to convert virtual addresses to core frame addresses. Selection of a page to be replaced may become more time consuming as the number of core frames increases. Furthermore, the percentage of space on a page devoted to overhead increases as the page size decreases. Actual design parameters for a virtual memory must consider the broader implementation constraints mentioned above.

The SAL Threshold Experiments

Since attempts to minimize hardware and overhead costs may require the use of only a few core frames, and hence large pages, it is desirable to find ways to increase the use ratio by reducing scattering. The second class of experiments produced comparative data on SYMBOL programs run using both the normal memory allocation strategy and an alternative strategy intended to reduce the scattering of memory references.

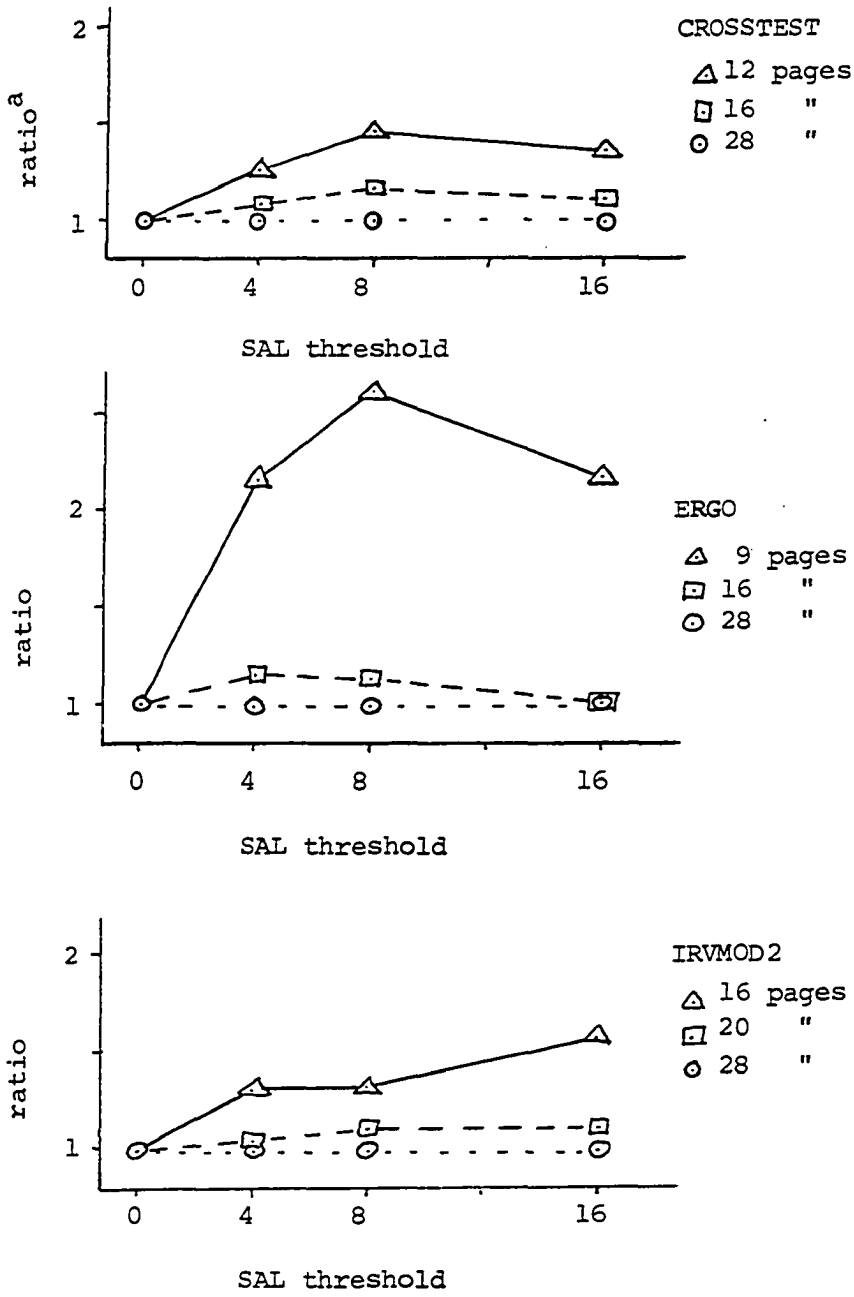
The algorithm for finding an available group is identical under the two strategies. The difference between them is the way the Space Available List (SAL) is built. In the alternate strategy, a page is not placed on the SAL unless it contains more than a specified number of available groups. This number is called the SAL threshold. The normal strategy uses a SAL threshold of zero.

The use of a nonzero threshold in building the Space Available List is an attempt to reduce scattering by reserving space on a page for intrapage expansion of logical strings, thereby reducing the interpage linking. If a page is not on the SAL, space can be allocated on that page only if the allocation request specifically references it. Such a request is made whenever a logical string is expanded. If at some point only a few groups on a page become available, it will not be put on the SAL, and hence, these groups will be effectively reserved for localized expansion. In this way, a string may tend to grow into pages it already occupies. Since the only memory allocation quantum in SYMBOL is the single group, an operation which requires that n groups be allocated causes a sequence of n requests for single groups. The standard memory management algorithm may cause these n groups to be spread over as many as n pages if the SAL is heavily populated with pages containing only one available group. By imposing a SAL threshold s, these groups are guaranteed to be found on the first n/s pages of the SAL. Hence when the n requests are all for groups in the same structure, this structure remains more compact.

As in the first class of experiments, each of the test programs was run as the only active task in the system once it had been brought to a fixed initial condition. Each program was run given a variety of combinations of twenty-eight group core frames and SAL thresholds. During each program's execution, the microprocessor again monitored the same four signals. In addition, at the end of program execution,

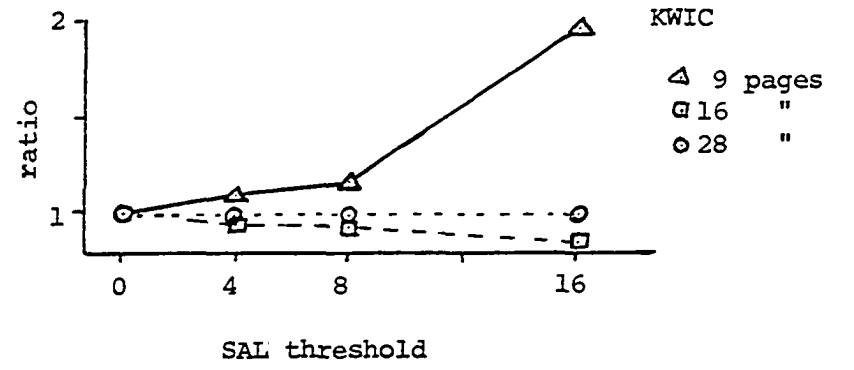
the SYMBOL programs NAME TABLE PAGESCAN and CONNECTIVITY were run to examine the test program's data space and gather data concerning its size and composition. The static and dynamic data for each of the test programs are summarized in Tables 5 through 10. The dynamic data again report the percentage of CP utilization, the execution time in seconds, the number of drum operations performed, and the percentage of CP shutdowns which occurred after more than 0, more than 256, and more than 1024 CP instructions had been fetched since the previous shutdown. The static data for each configuration consist of the number of pages used for data, the number of pages occupied by pieces of the program's major data structure, the number of page transitions needed to traverse that structure, and the average over all data space pages of the percentage of addresses each page contained which referred to other data space pages. The last three items of static data describe the program at its conclusion and cannot be guaranteed to be representative of earlier transient memory states, but instead illustrate the cumulative effects of the SAL threshold.

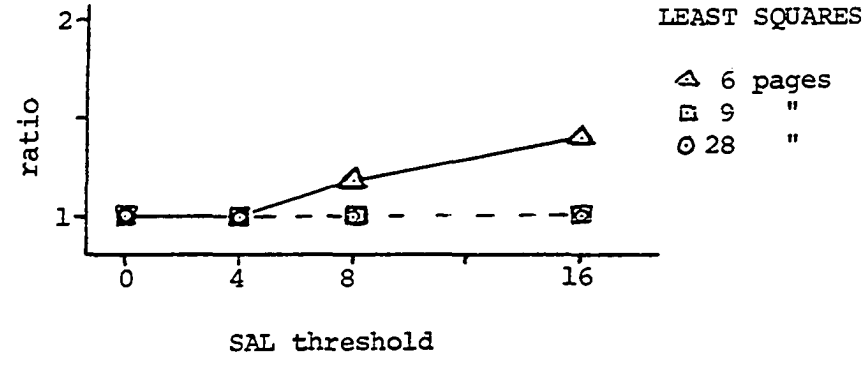
The graphs in Figure 6 show the effect of the SAL threshold on CP utilization. The entries are obtained by dividing the CP utilization obtained for p pages with SAL = s by that obtained for p pages with SAL = 0. These graphs and the data in Tables 5 through 10 show that for each of the test programs, if the core size is large enough that the program does little paging, then a nonzero SAL threshold has very little effect on peformance. However, if a substantial amount of paging is required under the normal allocation strategy (SAL = 0),



[a]ratio = (CP utilization for SAL = s)/(CP utilization for SAL = 0)

Figure 6. The effect of the SAL threshold on CP utilization





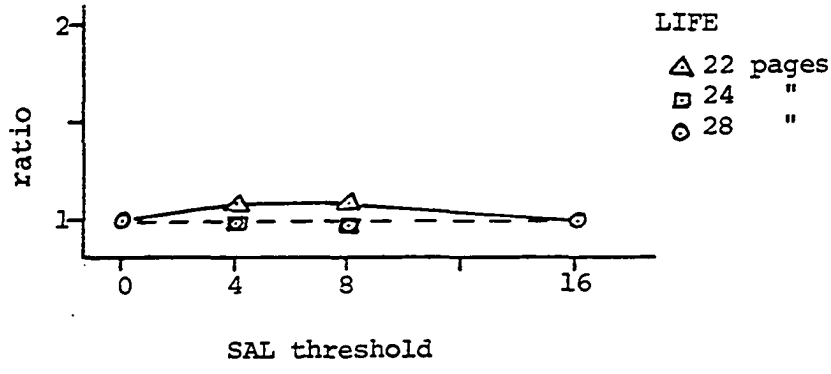
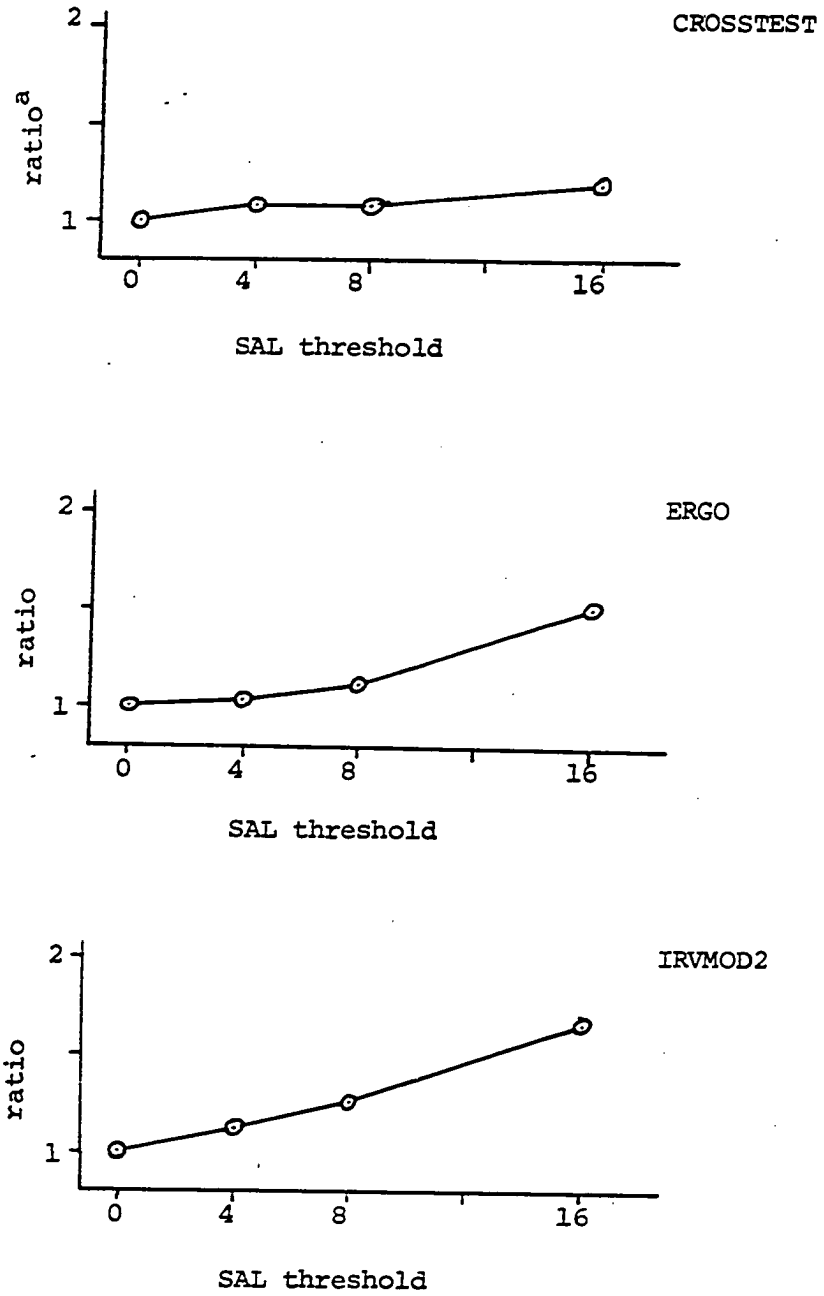


Figure 6. (continued)

then a significant increase in CP utilization is obtained by using SAL threshold values of eight and sixteen. This improved utilization is the result of performing fewer drum operations, and hence executing the program in less time. The reduced level of paging activity produces an upward shift in the distribution of the number of CP instructions fetched between shutdowns.

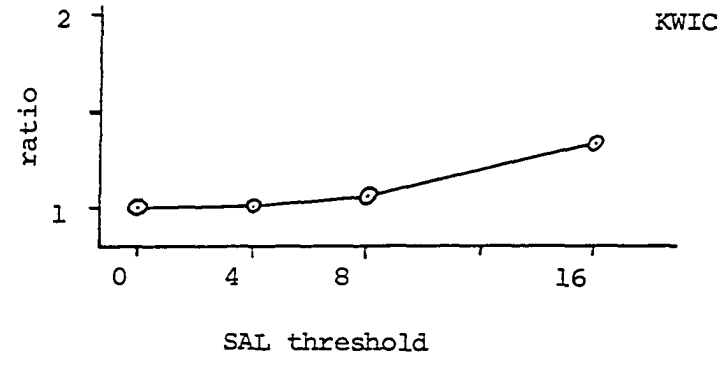
The static data displayed in Tables 5 through 10 indicate that the imposition of a SAL threshold can substantially reduce both the number of pages on which the components of a structure lie and the number of page transitions necessary to traverse the entire structure. The number of transitions in a traversal is extremely important because SYMBOL must follow links to access any specified component or substructure of a structure.

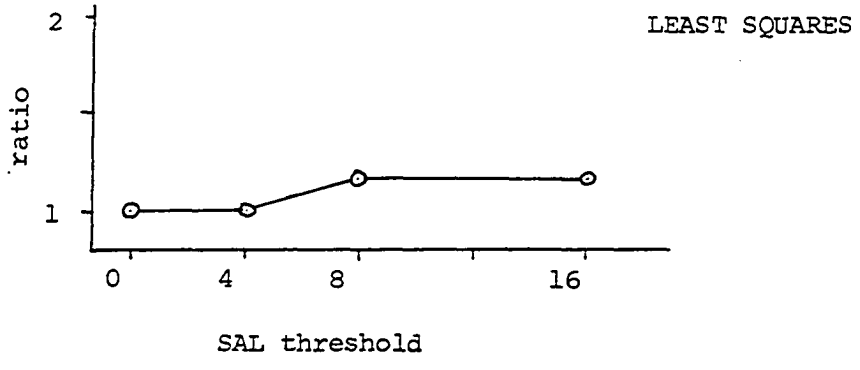
For the configurations using 28 core frames, the ratio of the number of pages used at SAL = s to the number used at SAL = 0 is plotted in Figure 7. The number of pages used in all configurations is shown in Tables 5 through 10. The experimental data clearly show that imposing a SAL threshold can force a program's data space to occupy a greater number of pages since there is then available space whose accessibility is restricted. Although the availability of extra backing store pages is usually not a problem in itself, too great an expansion of the programs's data space can have serious effects on performance. For instance, the program ERGO running with nine core frames requires 34 pages of data space for SAL = 0, 38 pages for SAL = 8, and 52 pages for SAL = 16 (an increase of approximately



[a]ratio = (pages used for SAL = s)/(pages used for SAL = 0)

Figure 7. The effect of the SAL threshold on pages used (28 core frames)





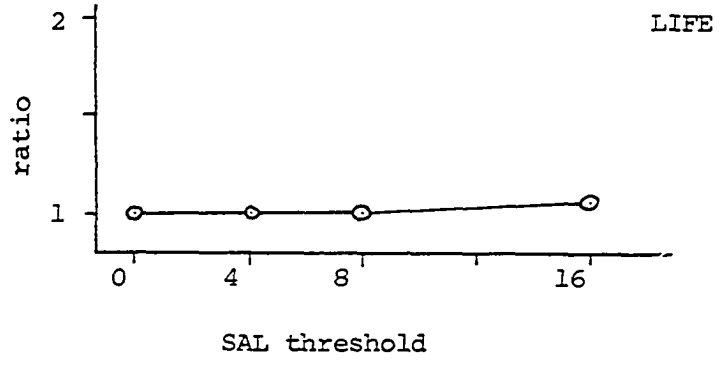


Figure 7. (continued)

50%). The increase in pages required from SAL = 8 to SAL = 16 must have been due to the fact that typically at least nine, but fewer than seventeen, groups were available on a page for reallocation. Assuming that the 34 pages used at SAL = 0 were compact, calculations show that approximately 9.7 groups per page were available. In fact, output from the CONNECTIVITY program shows that of the 44 pages with available groups which were not on the SAL, only three had fewer than nine groups available and 33 had exactly ten groups available. Since fewer than the threshold number of groups were available, large portions of these pages were simply wasted because ERGO did not expand existing strings, effectively forcing an upper bound on the use ratio of each page. Although growth into completely new pages was itself compact, the set of needed items fell on more pages, hence requiring more paging. Consequently the program could not attain the full benefit of the SAL threshold, and performance for SAL = 16, although better than for SAL = 0, was poorer than for SAL = 8. Another consequence of the wasted space which can result from a high SAL threshold is that certain programs are not able to obtain the minimum amount of core they need. For example, a particular assignment statement in LIFE requires very nearly twenty-two full core frames. Running LIFE with SAL = 16 restricts the use of space so that, given only twenty-two core frames, the statement cannot execute.

The effects of the SAL threshold were further tested by supplying several of the test programs with much larger sets of data. The

program CROSSTEST was given data which caused it to run for four hours under the normal configuration. By imposing a SAL threshold of eight, the execution time was reduced to two hours, and by imposing a threshold of sixteen, the running time was slightly longer than two hours. The program KWIC was given a list of 100 titles and authors to index, rather than the list of ten used for the earlier runs. Using 28 core frames, increasing the SAL threshold from 0 to 16 increased the CP utilization from 68.8% to 74.7% and decreased the number of drum operations performed from 10,684 to 7,876. Using nine full core frames, the improvements were even more pronounced. The CP utilization rose from 9.3% to 23.6%, and the number of drum operations fell from 245,527 to 78,995. In addition, the percentage of off-page references dropped from 61.3% to 7.5%. Thus we conclude that the benefits of the SAL threshold seen for small programs continue and are enhanced for large programs.

Since SYMBOL is capable of supporting multiprogramming, additional experiments involving several of the test programs were run to determine to what extent the benefits of the SAL threshold apply to a multiprogramming environment. Each of these programs was run on two terminals simultaneously, and the number of core frames used and the SAL threshold were varied as in the single terminal experiments. The cumulative behaviors of the programs were measured dynamically via the microprocessor system. The effects of the SAL threshold followed the same trends in the multiterminal experiments as they had in the single terminal runs. Using 28 full core frames, varying the SAL threshold

had no significant effect. For CROSSTEST, using 12 core frames and varying the SAL threshold from 0 to 8 decreased the execution time by 39% and the drum operations by 42%. These decreases were similar to those the SAL threshold produced for the same configurations in the single terminal experiments. For IRVMOD2, however, the imposition of a SAL threshold produced even greater improvements for the two terminal case than for the single terminal case. Using 20 core frames and varying the SAL threshold from 0 to 8 decreased the execution time by 29% and the drum operations by 43%.

Aside from the extra storage required for the data space, the use of a SAL threshold placed no other penalties on the system. No additional storage was required because, as explained earlier, the information needed to implement the SAL theshold was stored in each page's otherwise inactive initial group assignment counter. The updating and checking of this counter were performed at no time penalty with a small amount of additional hardware. Fewer list manipulations were performed since each allocation and freeing of a single group on an otherwise fully allocated page no longer caused deletions from and insertions to the SAL.

Table 5. Static and dynamic data for CROSSTEST

number of core frames	SAL threshold 0	4	8	16	
28	14	15	15	17	data pages
	10	11	11	12	structure pages
	88	68	70	52	traversal
	38.7	28.6	28.0	19.2	% off-page
	87.1	86.8	87.0	84.9	% CP busy
	36.9	37.1	37.0	37.6	seconds
	288	294	292	354	drum ops
	91.0	90.6	89.0	89.8	more than 0
	46.6	47.9	47.4	44.9	more than 256
	31.7	29.7	30.7	26.7	more than 1024
16	14	15	15	17	data pages
	10	11	11	11	structure pages
	72	75	67	54	traversal
	30.4	29.5	21.8	17.2	% off-page
	30.0	32.5	35.2	32.3	% CP busy
	107.2	99.2	91.6	99.8	seconds
	6070	5470	4910	5420	drum ops
	77.6	77.8	78.6	77.6	more than 0
	12.2	11.8	15.2	13.7	more than 256
	2.5	3.1	3.8	3.3	more than 1024
12	14	14	15	17	data pages
	9	10	10	13	structure pages
	75	78	59	58	traversal
	47.1	30.5	20.1	19.0	% off-page
	10.9	13.6	15.5	14.8	% CP busy
	297.5	235.7	207.9	217.1	seconds
	22120	16454	14276	15016	drum ops
	73.1	74.3	75.5	73.3	more than 0
	2.9	5.1	5.9	5.5	more than 256
	0.5	0.7	1.1	1.0	more than 1024

Table 6. Static and dynamic data for ERGO

number of core frames	SAL threshold 0	4	8	16	
28	34	35	38	51	data pages
	25	24	25	35	structure pages
	155	149	90	70	traversal
	17.8	13.5	8.5	3.6	% off-page
	97.5	96.7	96.7	94.3	% CP busy
	51.1	51.3	51.3	52.3	seconds
	128	133	152	224	drum ops
	93.7	92.9	92.5	93.0	more than 0
	76.8	76.5	73.8	65.0	more than 256
	63.2	64.3	61.7	62.2	more than 1024
16	34	36	38	52	data pages
	25	25	26	35	structure pages
	141	119	91	70	traversal
	16.4	12.4	7.0	3.5	% off-page
	79.2	93.9	91.2	78.8	% CP busy
	62.1	52.6	54.5	62.8	seconds
	1014	242	432	1124	drum ops
	80.2	86.4	78.9	81.3	more than 0
	38.9	58.5	53.3	57.2	more than 256
	22.6	51.2	37.6	14.4	more than 1024
9	34	36	38	52	data pages
	24	25	25	35	structure pages
	148	131	98	70	traversal
	17.5	12.3	9.0	3.7	% off-page
	22.9	49.4	59.9	50.2	% CP busy
	218.6	100.7	83.0	97.3	seconds
	13699	4163	2833	4001	drum ops
	59.3	61.8	70.6	66.4	more than 0
	7.7	18.4	27.0	24.0	more than 256
	1.2	7.8	9.1	5.1	more than 1024

Table 7. Static and dynamic data for IRVMOD2

number of core frames	SAL threshold 0	4	8	16	
28	16	18	20	27	data pages
	12	13	11	13	structure pages
	62	55	45	52	traversal
	36.0	32.5	25.7	19.5	% off-page
	95.9	96.9	96.5	94.7	% CP busy
	232.0	232.1	231.5	236.4	seconds
	522	624	626	1080	drum ops
	80.4	80.3	81.8	85.4	more than 0
	53.8	54.4	57.8	57.6	more than 256
	43.6	44.6	46.3	42.9	more than 1024
20	16	18	19	26	data pages
	12	11	9	14	structure pages
	62	47	36	44	traversal
	39.4	34.5	26.6	24.2	% off-page
	75.1	78.0	82.7	82.6	% CP busy
	300.3	287.8	270.6	269.3	seconds
	6164	5266	3892	3890	drum ops
	83.5	87.1	83.8	86.7	more than 0
	31.5	39.6	42.2	45.6	more than 256
	15.3	20.7	27.1	29.9	more than 1024
16	16	18	20	26	data pages
	13	13	10	11	structure pages
	62	45	45	43	traversal
	41.2	40.3	27.3	20.7	% off-page
	33.9	45.9	45.0	53.6	% CP busy
	674.5	486.9	495.0	415.5	seconds
	28804	21472	22428	15784	drum ops
	82.0	83.2	84.3	84.4	more than 0
	18.2	26.2	27.0	28.9	more than 256
	3.6	7.4	7.1	12.1	more than 1024

Table 8. Static and dynamic data for KWIC

number of core frames	SAL threshold 0	4	8	16	
28	15	16	16	20	data pages
	12	13	13	17	structure pages
	75	66	70	61	traversal
	22.3	18.9	19.3	9.4	% off-page
	96.9	96.9	97.2	96.9	% CP busy
	36.1	35.9	35.8	35.9	seconds
	22	22	23	26	drum ops
	78.8	78.8	85.3	81.1	more than 0
	51.5	51.5	52.9	56.8	more than 256
	48.5	48.5	47.1	51.4	more than 1024
16	15	15	17	19	data pages
	13	13	14	16	structure pages
	75	67	66	61	traversal
	22.4	19.6	18.1	14.6	% off-page
	95.3	93.3	91.8	84.7	% CP busy
	36.6	37.5	37.9	41.3	seconds
	72	166	200	474	drum ops
	81.8	85.2	87.4	85.7	more than 0
	40.0	42.0	44.5	38.6	more than 256
	36.4	34.1	35.3	29.0	more than 1024
9	15	16	16	19	data pages
	12	12	13	15	structure pages
	75	65	67	63	traversal
	21.5	14.2	15.1	14.4	% off-page
	18.1	20.8	21.6	36.0	% CP busy
	193.2	168.0	162.3	97.3	seconds
	13020	11027	10323	4931	drum ops
	73.4	71.4	68.6	69.9	more than 0
	10.4	12.0	11.4	16.3	more than 256
	2.3	2.6	2.9	7.6	more than 1024

Table 9. Static and dynamic data for LEAST SQUARES

number of core frames	SAL threshold 0	4	8	16	
28	6	6	7	7	data pages
	1	2	2	1	structure pages
	1	5	6	1	traversal
	22.1	13.8	15.8	19.6	% off-page
	97.9	98.4	98.9	97.9	% CP busy
	9.4	9.4	9.4	9.4	seconds
	11	11	12	12	drum ops
	83.3	83.3	84.2	84.2	more than 0
	61.1	55.6	57.9	57.9	more than 256
	50.0	44.4	42.1	42.1	more than 1024
9	6	6	6	7	data pages
	2	1	3	1	structure pages
	6	1	5	1	traversal
	23.3	17.2	24.2	16.2	% off-page
	96.9	96.9	96.9	95.8	% CP busy
	9.6	9.6	9.6	9.6	seconds
	34	25	25	25	drum ops
	83.3	83.3	83.3	84.0	more than 0
	41.6	41.6	41.6	48.0	more than 256
	33.3	33.3	33.3	32.0	more than 1024
6	5	5	5	6	data pages
	3	2	1	2	structure pages
	8	5	1	2	traversal
	25.0	21.1	16.4	14.7	% off-page
	47.2	46.7	57.0	66.4	% CP busy
	19.7	19.5	16.5	14.0	seconds
	840	848	608	444	drum ops
	75.4	71.0	61.1	93.5	more than 0
	4.6	5.7	3.8	4.7	more than 256
	1.6	1.4	2.5	3.4	more than 1024

Table 10. Static and dynamic data for LIFE

number of core frames	SAL threshold 0	4	8	16	
28	35	36	36	37	data pages
	11	10	10	10	structure pages
	31	26	26	29	traversal
	13.2	7.1	7.3	9.5	% off-page
	88.6	89.8	89.1	84.6	% CP busy
	45.5	45.1	45.0	47.3	seconds
	413	408	374	545	drum ops
	75.8	75.5	76.1	78.9	more than 0
	47.1	45.1	49.5	50.8	more than 256
	42.6	44.3	39.2	41.9	more than 1024
24	35	36	36	37	data pages
	11	10	10	10	structure pages
	31	26	26	29	traversal
	10.7	7.5	7.4	6.6	% off-page
	82.3	83.8	82.7	80.0	% CP busy
	49.6	49.3	48.6	50.6	seconds
	764	747	724	840	drum ops
	72.8	78.0	75.1	75.2	more than 0
	41.3	46.6	45.5	46.6	more than 256
	35.8	40.7	40.6	36.7	more than 1024
22	35	36	36	won't run	data pages
	10	10	10		structure pages
	31	28	28		traversal
	10.1	7.3	7.5		% off-page
	70.7	74.7	74.5		% CP busy
	57.0	55.3	56.0		seconds
	1223	1122	1204		drum ops
	64.6	67.9	69.0		more than 0
	33.5	39.4	38.9		more than 256
	26.6	31.4	27.1		more than 1024

CONCLUSIONS

Previous research on the effect of page size on system performance (4a,5b,13b) has not dealt with systems which manage dynamic storage. It is reasonable to expect that such systems produce greater scattering of memory references than those which manage only static storage and hence may derive even greater benefit from the use of small pages. The results of the experiments performed on SYMBOL lend support to this hypothesis. In a number of instances, it was determined that the page size, and not only the memory size, was crucial in determining whether a given program could execute. Programs were able to execute given a fixed amount of core memory configured as a large number of small pages, but were unable to execute when this same amount of memory was configured as a small number of large pages. An extremely poor use ratio for the large pages would explain this behavior. Too much of the core memory was occupied by unreferenced items, leaving an inadequate amount of working space for the program. In addition, some of the earlier work on paging in a static environment has indicated that page sizes can be reduced too much, in the sense that performance begins to degrade again for very small pages. Such trends were not evident in SYMBOL. On the contrary, in all cases for which data were obtained, as the page size was reduced, the number of page faults was also reduced. Hence, the scattering in SYMBOL must be so severe that the use ratio continues to improve as the page size is reduced toward a minimum.

The use of the SAL threshold in SYMBOL's memory allocation strategy is similar in philosophy to a strategy used in an implementation of LISP (4c). Both strategies were able to enhance the performance of their respective systems. The use of the SAL threshold in SYMBOL is more general in that it applies to all storage allocation, not just that caused by one particular language operation (the CONS operator in LISP). The experimental results indicate that use of the threshold produces very little degradation in programs which have sufficient memory while producing dramatic improvements in the execution of programs which otherwise need to do a significant amount of paging. The data obtained by the CONNECTIVITY program show that the the SAL threshold was highly successful in reducing the number of off-page references contained on a page, and hence diminished the scattering of memory references. These results are most gratifying, especially in view of the fact that the SAL threshold is so easy to implement on SYMBOL. Allocation mechanisms with similar strategies should be simple to implement on other systems as well.

The results presented here indicate that a very small page size might yield reasonable performance in SYMBOL. If pages were very small, perhaps only one group long, there would obviously be no need for the SAL threshold. The choice of a page size is, of course, dependent upon technologies. When SYMBOL was designed, the access time for the available backing store was several thousand times greater than the access time for core memory. Thus it seemed worthwhile to bring a large number of groups into core whenever the

drum was accessed. As technologies change and speeds improve, such rationales may disappear. However, until that time, pages will probably continue to be reasonably large, and memory allocation strategies such as the SAL threshold will need to be implemented to control scattering.

The data presented in this paper demonstrate that the effects of scattering can be reduced by the use of small pages with no change in the memory allocation strategy, and that scattering itself can be curbed by using a sophisticated memory allocation strategy, such as that associated with the SAL threshold. Improvements in performance which resulted in doubling the CP utilization and cutting the number of drum operations in half were not uncommon for either experiment. Each of these approaches to controlling scattering has advantages over program restructuring. Restructuring techniques are applicable only to static items, while these methods can be applied to both static and dynamic storage. Furthermore, restructuring is an iterative process and, as such, is of no benefit to programs which are run only a very few times or to those written by naive users whose lack of expertise precludes effective restructuring. On the other hand, the methods presented here apply to a program which is executed even if only once, whether it was written by a novice or an expert.

This research has produced one other significant result, although not one related to the topic under investigation. It has demonstrated that a modest microprocessor system can be a valuable tool in the study of computer system performance. Such a microprocessor system

can monitor, record, and display enough information to give a useful profile of system performance and is available at a very affordable price.

The results discussed in this paper were obtained by measuring a set of diverse test programs with data sets of vastly different sizes. Whenever practical, data were taken for each program running alone and on two terminals simultaneously. The results obtained show the same behavior trends for each program. For these reasons it is believed that the results are indicative of the behavior of all SYMBOL programs. It is further believed that the general principles discussed here apply to scattering in other virtual memory systems as well.

SUGGESTIONS FOR FUTURE WORK

Within SYMBOL, scattering is reduced by segregating pages into usage classes and maintaining a list of the pages which belong to each class. The experiments performed suggest that performance enhancement could be had by breaking the data usage class down into several subclasses, each with its own page list. The system might automatically maintain subclasses for stacks, temporaries, and structures. In addition, the user might be given the option of specifying sets of identifiers whose values should be given their own page lists, thus using his knowledge of the program's data accessing patterns to reduce scattering. In a similar vein, the user might be allowed to specify the shape and primary mode of access (e.g., forward sequential, backward sequential, random) of a large structure whose shape is static. The system could then use this information to allocate storage for that structure compactly and in a way which would promote its efficient access according to the mode specified.

It would be valuable to determine to what extent the use of smaller pages would benefit other virtual memory systems which produce varying degrees of scattering. Studies could be undertaken to compare the benefits of program restructuring and the use of small pages. It is suspected that small pages may give greater improvements in performance because program restructuring depends on static relationships and accessing patterns of data.

If experiments show that large pages are sometimes preferable,

the benefits of large pages might be obtainable while still using small pages by replacing several core frames with contiguous pages from the backing store at page fault time. To extend the work of (2), such a system should be able to vary the number of core frames replaced at a single page fault so that it may, in effect, use a page size suited to the locality of the program being executed.

The experimental data seem to imply that there is no single value for the SAL threshold which guarantees a maximum increase in performance for all programs. The optimal threshold value probably varies during the lifetime of a program. Hence, some means of deciding upon the optimal value and resetting the threshold during execution would be desirable.

Memory allocation strategies similar to the SAL threshold strategy should be implementable on other virtual memory systems. It would be beneficial to see if such strategies give improvements in performance similar to those obtained on SYMBOL, and at what costs any such improvements are obtained.

In virtual memory systems it is useful to avoid references across page boundaries because of the costs of performing a paging operation. Likewise, for systems with moving arm disks it is useful to avoid references across cylinder boundaries to avoid the delays associated with arm motions. SYMBOL's memory allocation strategies, particularly its use of page lists, available group lists, and the newly implemented SAL threshold, should be adaptable to efficient allocation of disk storage and may be found to reduce the scattering of pages on

a disk just as they have been able to reduce scattering within SYMBOL programs.

BIBLIOGRAPHY

1. Baecker, H. D. "Some Notes on Dynamic Storage Allocation." In Virtual Storage, pp. 193-202. Edited by Infotech International Limited. Maidenhead , England: Nicholson House, 1976.

2. Baer, Jean-Loup, and Sager, Gary R. "Dynamic Improvement of Locality in Virtual Memory Systems." IEEE Transactions on Software Engineering SE-2 (March 1976): 54-62.

3. Batson, Alan. "Program Behavior at the Symbolic Level." IEEE Computer 9 (November 1976): 21-26.

4a. Belady, L. A. "A study of replacement algorithms for a virtual-storage computer." IBM Systems Journal 5, No. 2 (1966): 78-101.

4b. Belady, L.A., and Kuehner, C. J. "Dynamic Space-Sharing in Computer Systems." CACM 12 (May 1969): 282-288.

4c. Bobrow, Daniel G., and Murphy, Daniel L. "Structure of a LISP System Using Two-Level Storage." CACM 10 (March 1967): 155-159.

4d. Boyse, John W. "Execution Characteristics of Programs in a Page-on-Demand System." CACM 17 (April 1974): 192-196.

5a. Brundage, Robert E., and Batson, Alan P. "Computational Processor Demands of Algol-60 Programs." ACM Operating Systems Review 9, No. 5 (1975): 161-168.

5b. Chu, Wesley W., and Opderbeck, Holger. "Performance of Replacement Algorithms with Different Page Sizes." IEEE Computer 7 (November 1974): 14-21.

6. Chu, Wesley, W., and Opderbeck, Holger. "Program Behavior and the Page-Fault-Frequency Replacement Algorithm." IEEE Computer 9 (November 1976): 29-38.

7. Clark, Douglas W. "An Empirical Study of List Structures in LISP." CACM 20 (February 1977): 78-87.

8. Comfort, W. T. "Multiword List Items." CACM 7 (June 1964): 357-362.

9. Denning, P. J. "The Working Set Model for Program Behavior." CACM 11 (May 1968): 323-333.

10a. Denning, P. J. "Virtual Memory." Computing Surveys 2, No. 3, (1970): 153-189.

10b. Denning, P. J.; Kahn, K. C.; Leroudier, J.; Potier, D.; and Suri, R. "Optimal Multiprogramming." Acta Informatica 7, No. 2 (1966): 197-216.

11. Ferrari, Domenico. "Improving Locality by Critical Working Sets." CACM 17 (November 1974): 614-620.

12. Ferrari, Domenico. "The Improvement of Program Behavior." IEEE Computer 9 (November 1976): 39-47.

13a. Gifford, David K. "Hardware Estimation of a Process' Primary Memory Requirements." CACM 20 (September 1977): 655-663.

13b. Hatfield, D. J. "Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance." IBM Journal of Research and Development 10 (January 1972): 58-66.

14. Hatfield, D. J., and Gerald, J. "Program restructuring for virtual memory." IBM Systems Journal 10, No. 3 (1971): 168-192.

15. Kilburn, T.; Edwards, D. B. G.; Lanigan, M. J.; and Sumner, F. H. "One-Level Storage System." IRE Transactions on Electronic Computers EC-11 (April 1962): 223-235.

16. Madison, A. Wayne, and Batson, Alan P. "Characteristics of Program Localities." CACM 19 (May 1976): 285-294.

17. Pittman, Tom. TINY BASIC. Unpublished program. San Jose, California: Itty Bitty Computers, 1976.

18. Prieve, Barton G., and Fabry, R. S. "VMIN -- An Optimal Variable-Space Page Replacement Algorithm." CACM 19 (May 1976): 295-297.

19. Richards, H., Jr. "SYMBOL II-R Language Reference Manual." Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, 1971.

20. Richards, H., Jr., and Zingg, R. J. "The Logical Structure of the Memory Resource in the SYMBOL 2R Computer." ACM-IEEE Symposium on High-Level-Language Computer Architecture. New York: Association for Computing Machinery, 1973.

21. Smith, William R.; Rice, Rex; Chesley, Gilman D.; Laliotis, Theodore A.; Lundstrom, Stephen F.; Calhoun, Myron A.; Gerould, Lawrence D.; and Cook, Thomas G. "SYMBOL -- a Large Experimental System Exploring Major Hardware Replacement of Software." AFIPS Spring Joint Computer Conference Proceedings 38 (1971): 601-616.

22. Spirn, Jeffrey R. Program Behavior: Models and Measurements. New York: Elsevier, 1977.

23. Svobodova, Liba. Computer Performance Measurement and Evaluation Methods: Analysis and Applications. New York: Elsevier, 1976.

24. Zingg, R. J., and Richards, H. Jr. "Symbol: A System Tailored to the Structure of Data." Proceedings of the National Electronics Conference 27 (1972): 306-311.

ACKNOWLEDGMENTS

The author wishes to thank the many individuals who have provided him with advice, encouragement, and assistance. Special thanks are due to Robert M. Stewart, his major professor and former head of the SYMBOL project, and to Roy J. Zingg, the present director of the project. The author is grateful to Perry Hutchison for his careful inspection of the manuscript and for his efforts in keeping SYMBOL in working order. He is also grateful to Dave Ditzel who offered many helpful suggestions and wrote a text editor for SYMBOL which has been of invaluable assistance in the preparation of the manuscript, from rough draft to final copy. The author is indebted to Bob Cmelik, Irv Hentzel, Ron Wolf, and several unknown programmers who wrote the programs upon which his measurements were taken.

Support for this research has been provided by a research assistantship from the Graduate College of Iowa State University and by the National Science Foundation under grant number GJ33097X.

Thanks are due to Mrs. LaDena Bishop of the University Library Thesis Office and her staff both for their patient and helpful advice in the preparation of the tables and figures and for their diligent checking of the manuscript.

Lastly the author wishes to thank his wife, Kathy, who has spent uncounted hours listening to and helping him clarify his ideas and also attending to the mechanical details of producing the manuscript. She also provided much needed support and encouragement.